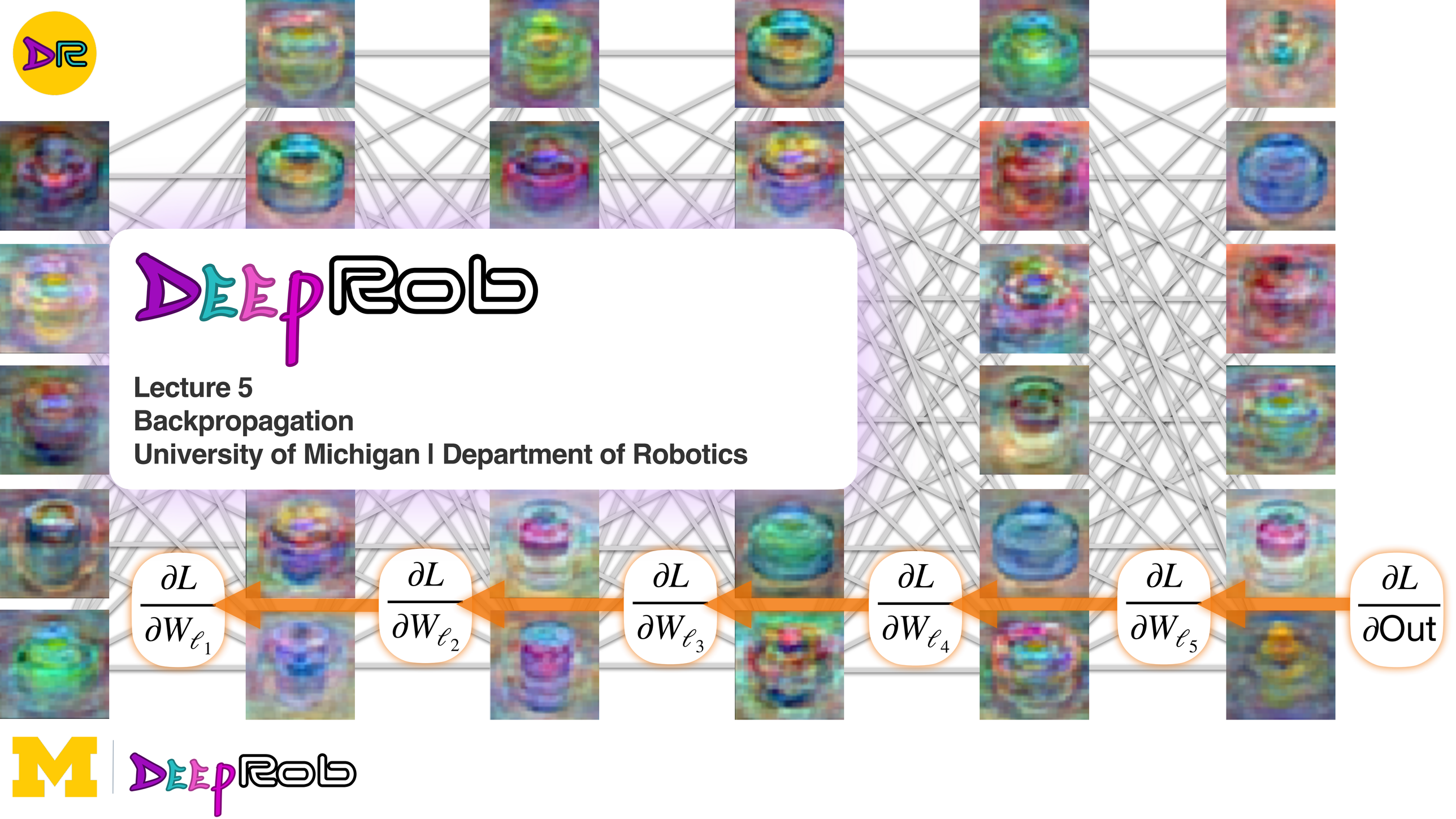
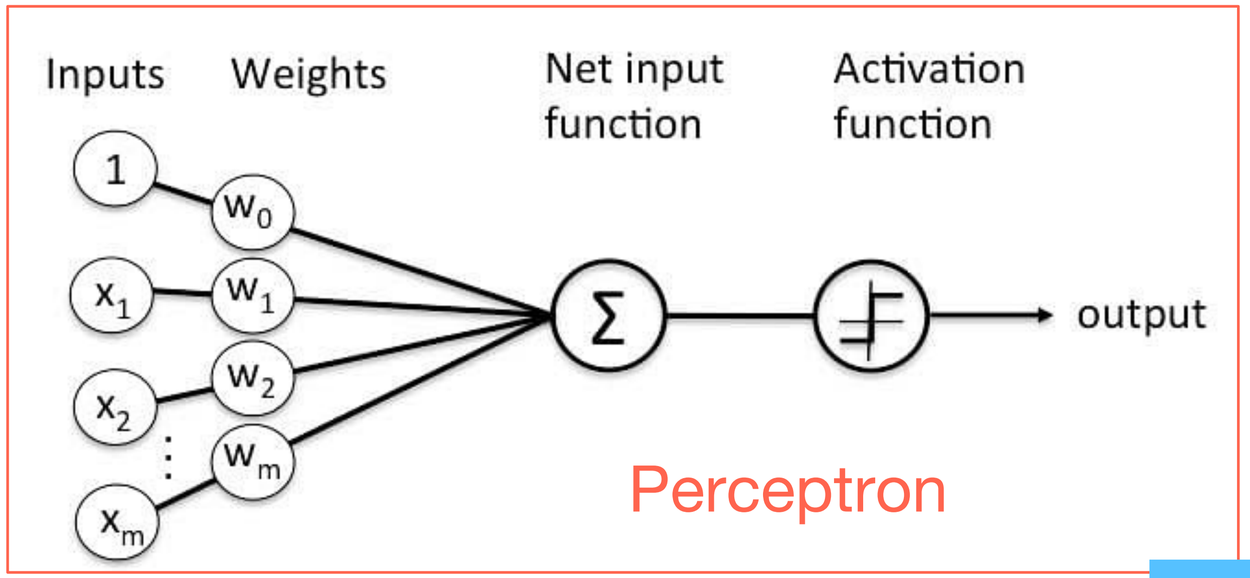

DEEPROB
Lecture 5
Backpropagation
University of Michigan | Department of Robotics
$\frac{\partial L}{\partial W_{\ell_1}}$
$\frac{\partial L}{\partial W_{\ell_2}}$
$\frac{\partial L}{\partial W_{\ell_3}}$
$\frac{\partial L}{\partial W_{\ell_4}}$
$\frac{\partial L}{\partial W_{\ell_5}}$
$\frac{\partial L}{\partial \text{Out}}$
DEEPROB

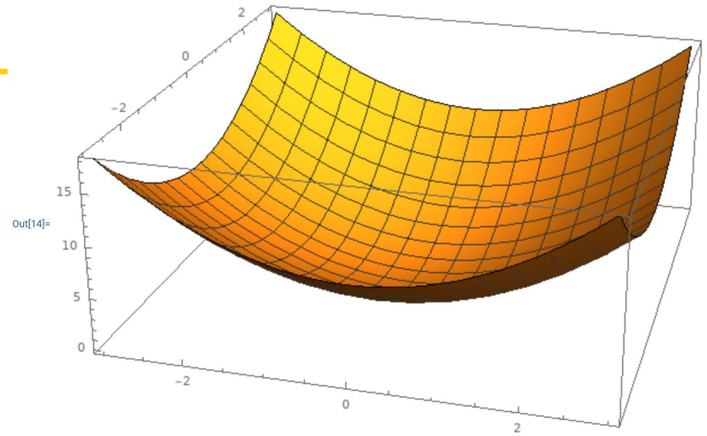
Project 1—Reminder

- Instructions and code available on the website
  - Here: deeprob.org/projects/project1/
- Implement KNN, linear classifier, and fully connected NN
- **Due Thursday, Feb.1, 11:59 PM EST**
- **Late policy: 3 late tokens (24hrs each with no penalty); 25% deduction for every day the submission was late after using all three late tokens**

Recap: Neural Networks

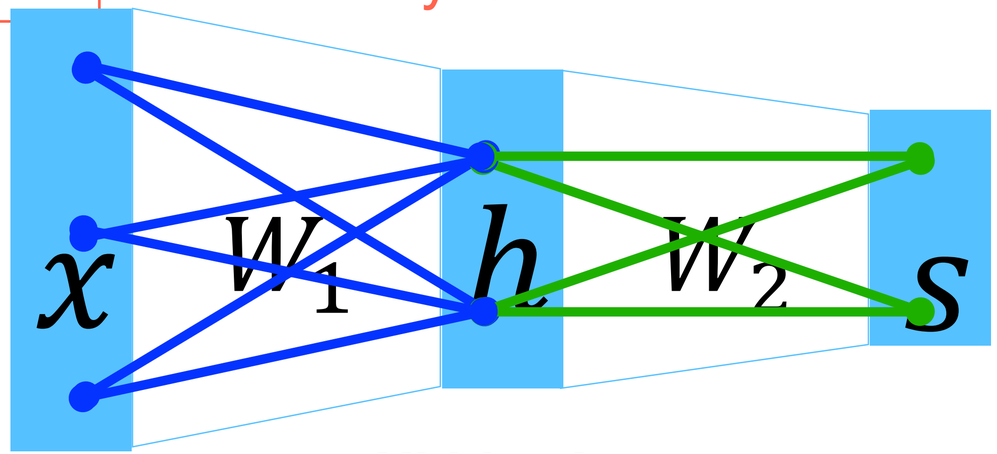


Perceptron



MLP

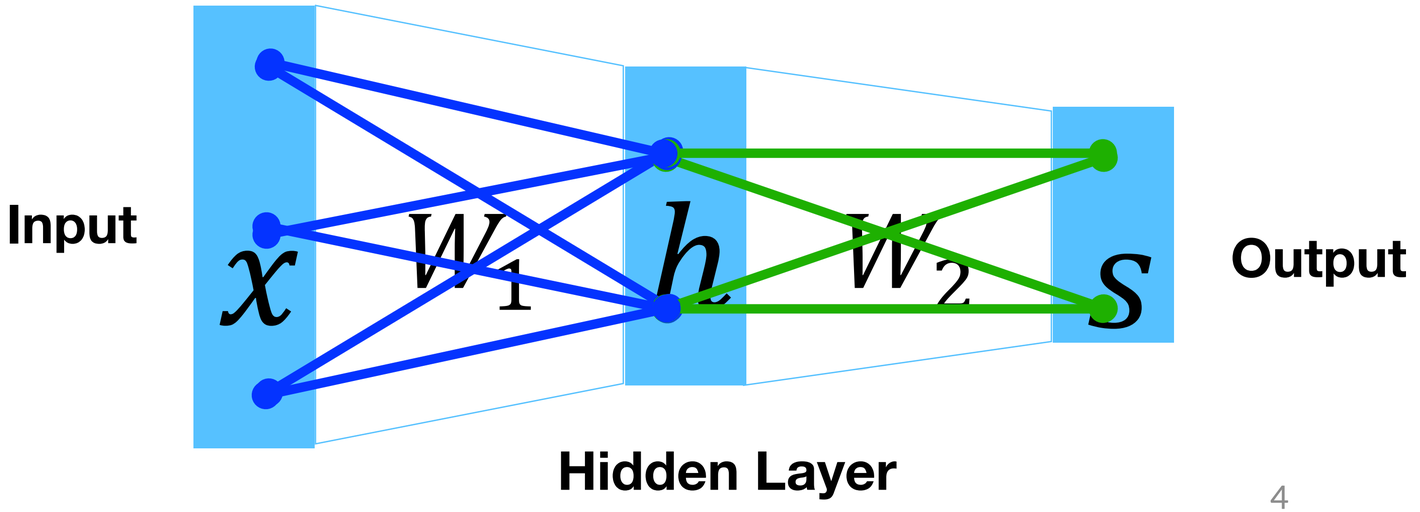
"Fully Connected"



Problem: How to compute gradients?

$$s = W_2 max(0, W_1 x + b_1) + b_2$$

Nonlinear score function



Problem: How to compute gradients?

$$s = W_2 max(0, W_1 x + b_1) + b_2$$ Nonlinear score function

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$ Per-element data loss

$$R(W) = \sum_k W_k^2$$ L2 regularization

$$L(W_1, W_2, b_1, b_2) = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2)$$ Total loss

If we can compute $\frac{\delta L}{\delta W_1}, \frac{\delta L}{\delta W_2}, \frac{\delta L}{\delta b_1}, \frac{\delta L}{\delta b_2}$ then we can optimize with SGD

(Bad) Idea: Derive $\nabla_W L$ on paper

$$s = f(x; W) = Wx$$

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

$$= \sum_{j \neq y_i} max(0, W_{j,:}x - W_{y_i,:}x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} max(0, W_{j,:}x - W_{y_i,:}x + 1) + \lambda \sum_k W_k^2$$

$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} max(0, W_{j,:}x - W_{y_i,:}x + 1) + \lambda \sum_k W_k^2 \right)$$

Problem: Very tedious with lots of matrix calculus

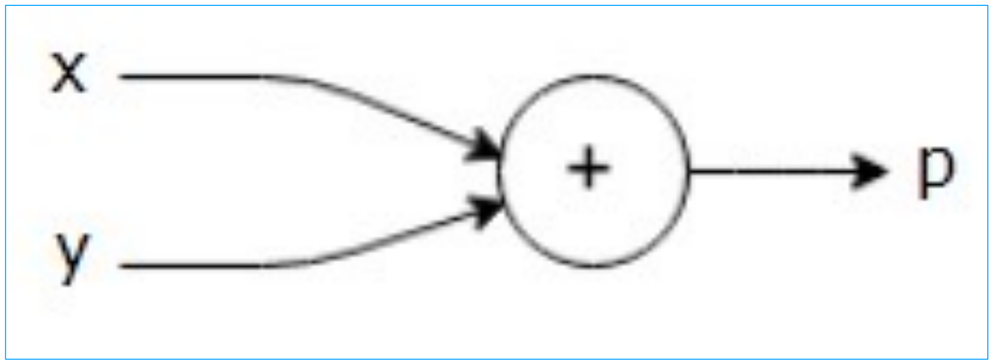
Problem: What if we want to change the loss? E.g. use softmax instead of SVM? Need to re-derive from scratch. Not modular!

Problem: Not feasible for very complex models!

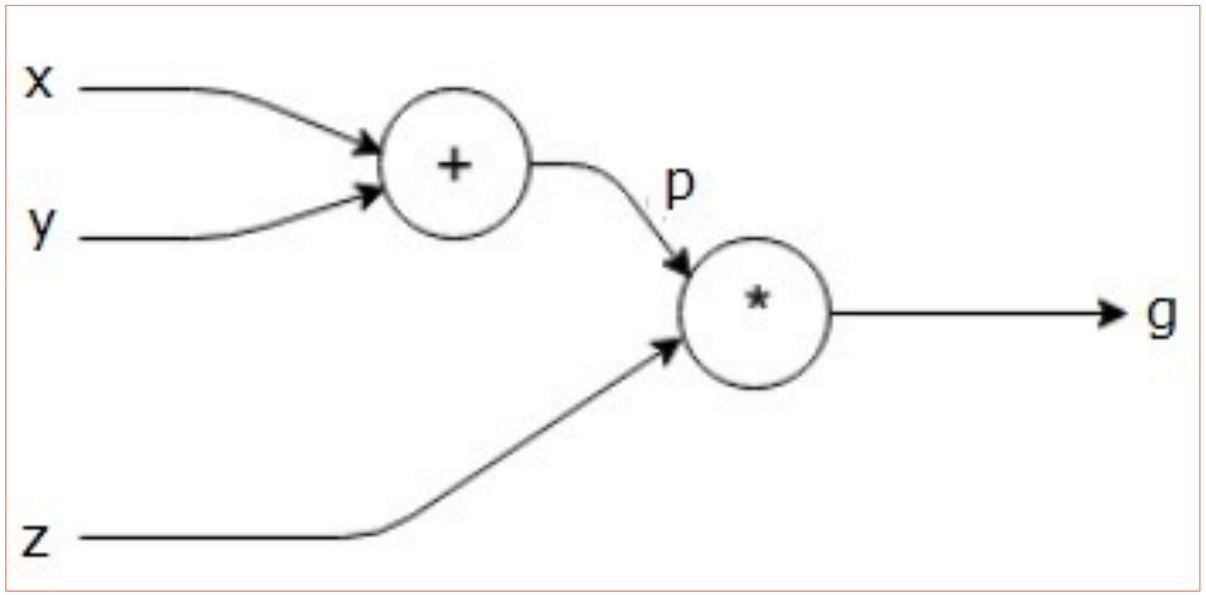
Better Idea: Computational Graphs

Simple example:

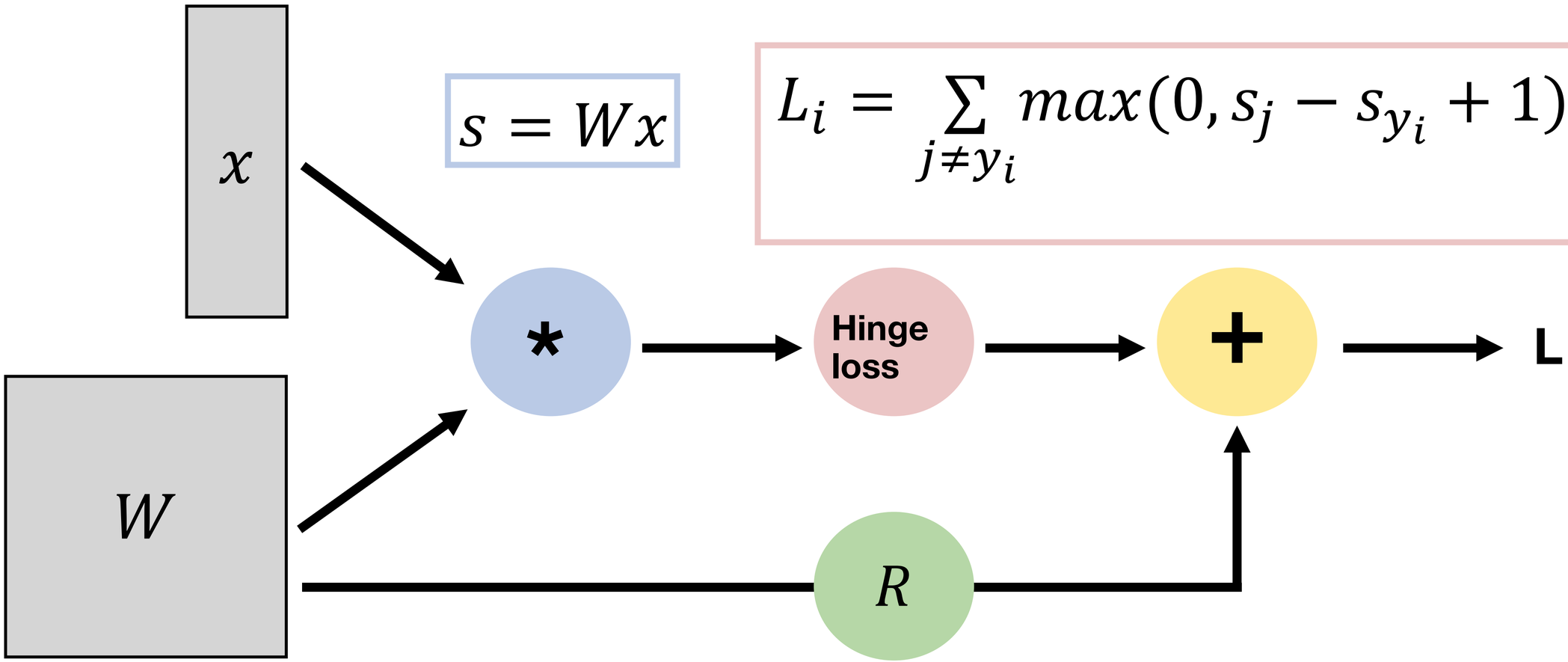
$$p = x + y$$



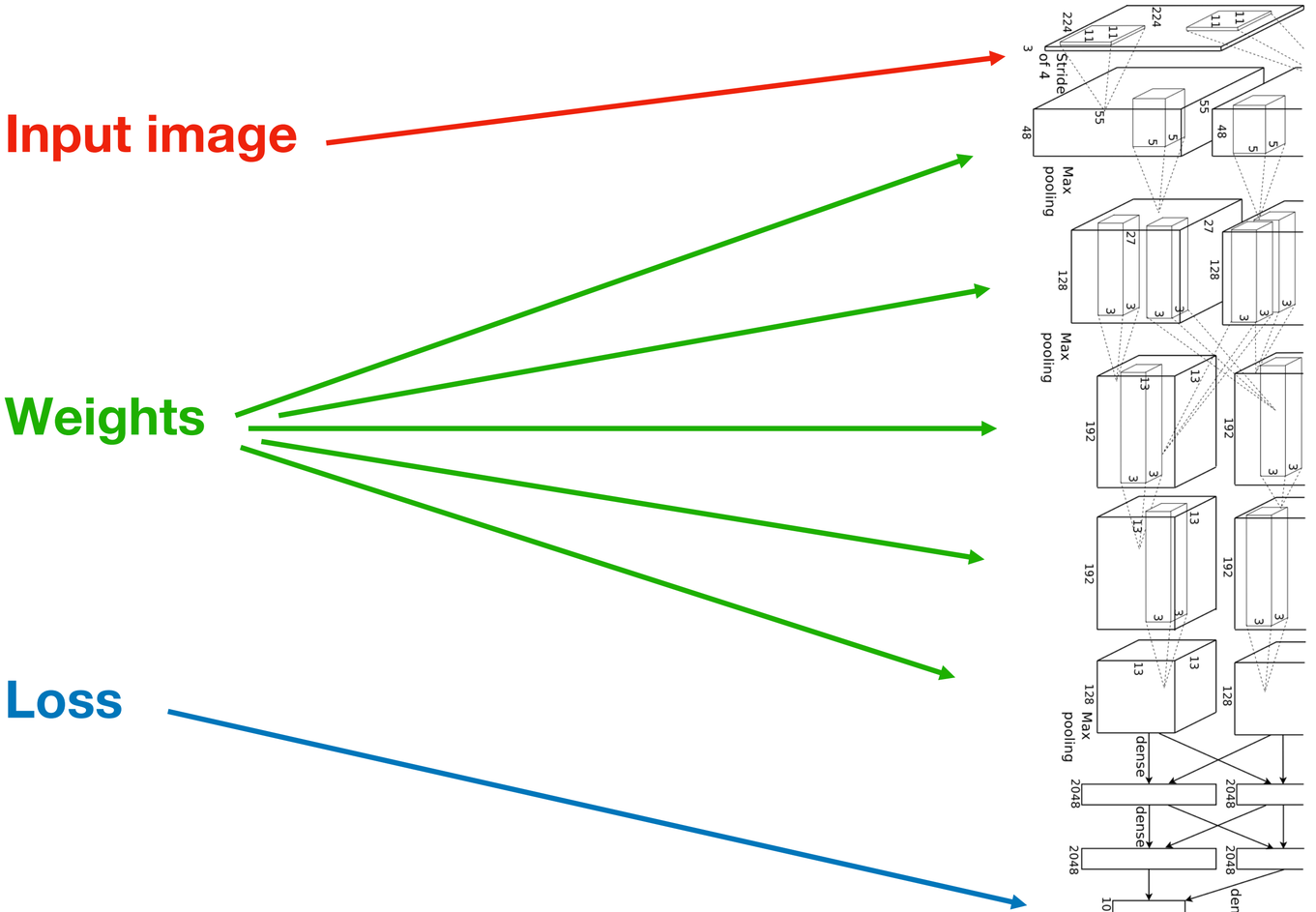
$$p = (x + y) * z$$



Better Idea: Computational Graphs

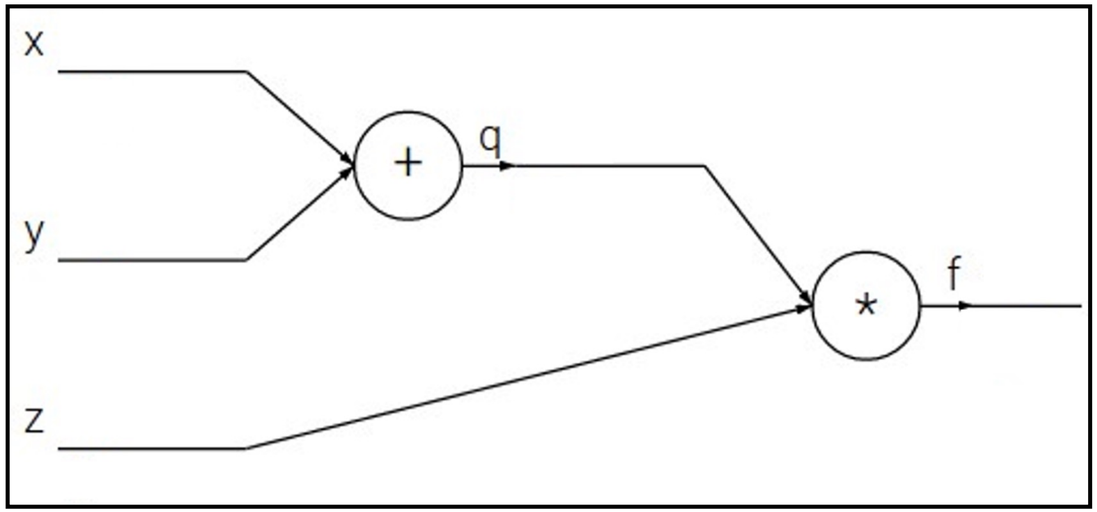


Deep Network (AlexNet)



Backpropagation: Simple Example

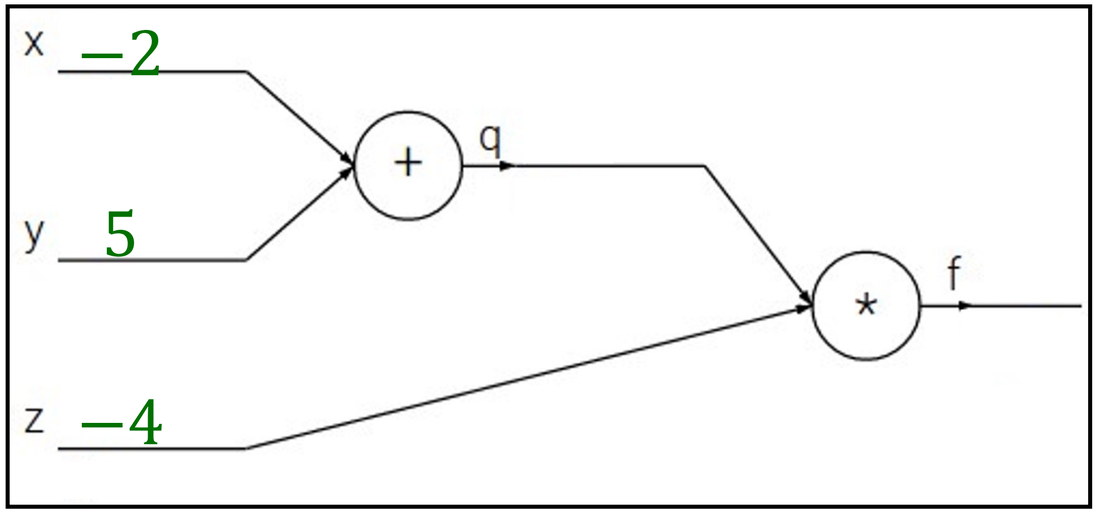
$$f(x, y, z) = (x + y) \cdot z$$



Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

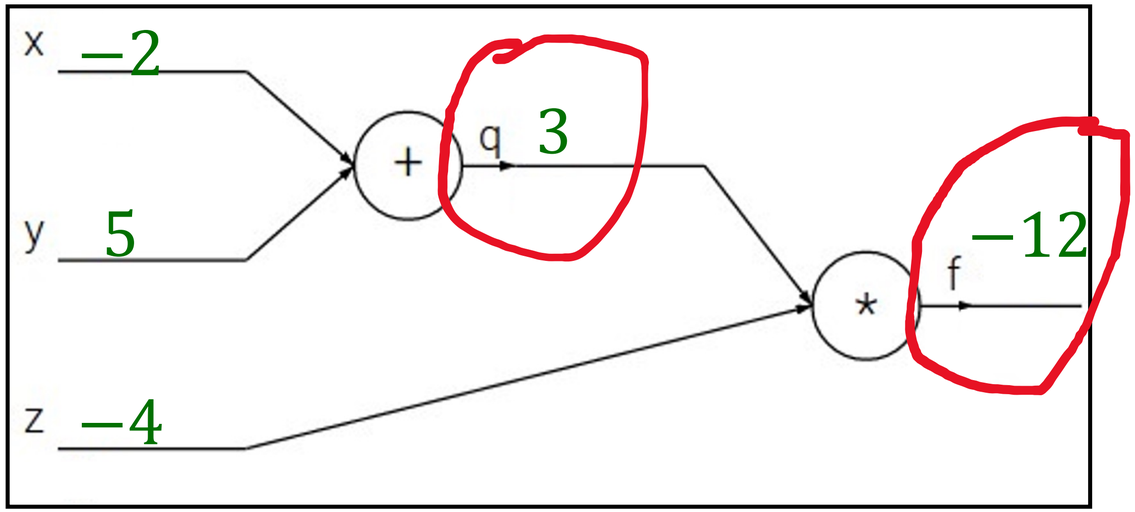
e.g. $x = -2, y = 5, z = -4$



Backpropagation: Forward Pass

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. Forward pass: Compute outputs

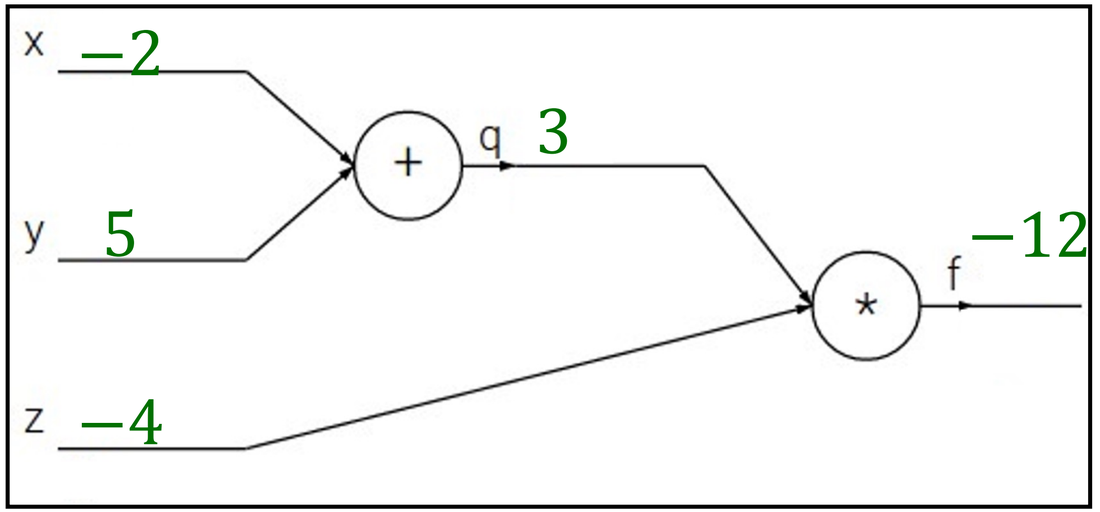
$$q = x + y = -2 + 5 = 3$$

$$f = q \cdot z = 3 * (-4) = -12$$

Backpropagation: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. **Forward pass**: Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. **Backward pass**: Compute derivatives

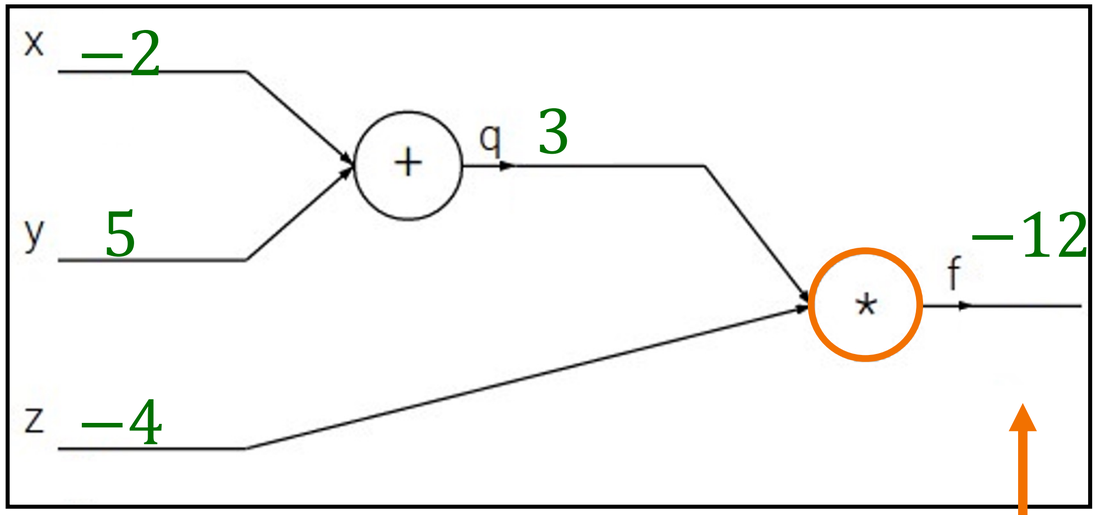
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Problem: What is $\frac{\partial f}{\partial f}$?

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. Forward pass: Compute outputs

$q = x + y$ $\quad$ $f = q \cdot z$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

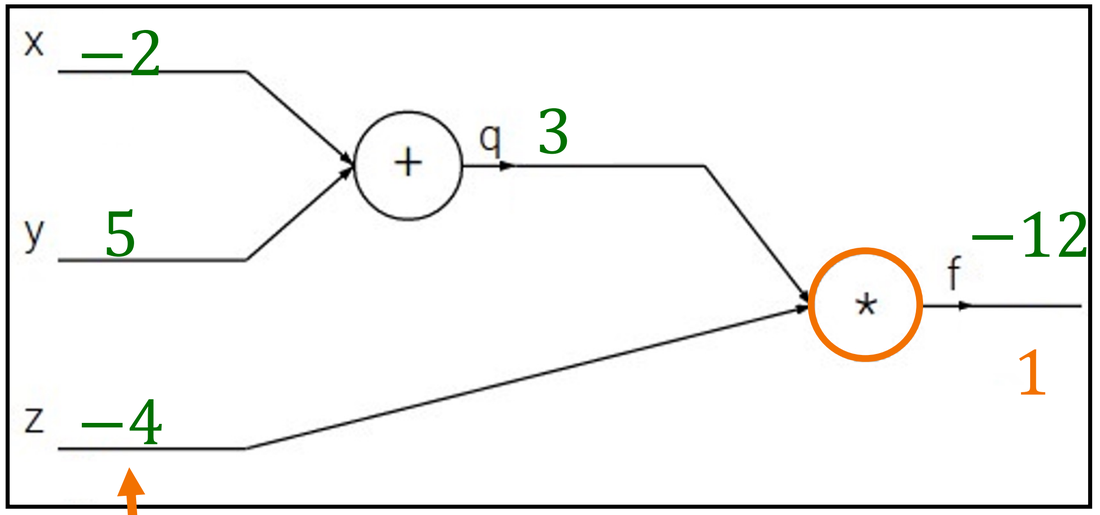
$\frac{\partial f}{\partial f}$

$= 1$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. Forward pass: Compute outputs

$q = x + y$ $\quad$ $f = q \cdot z$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

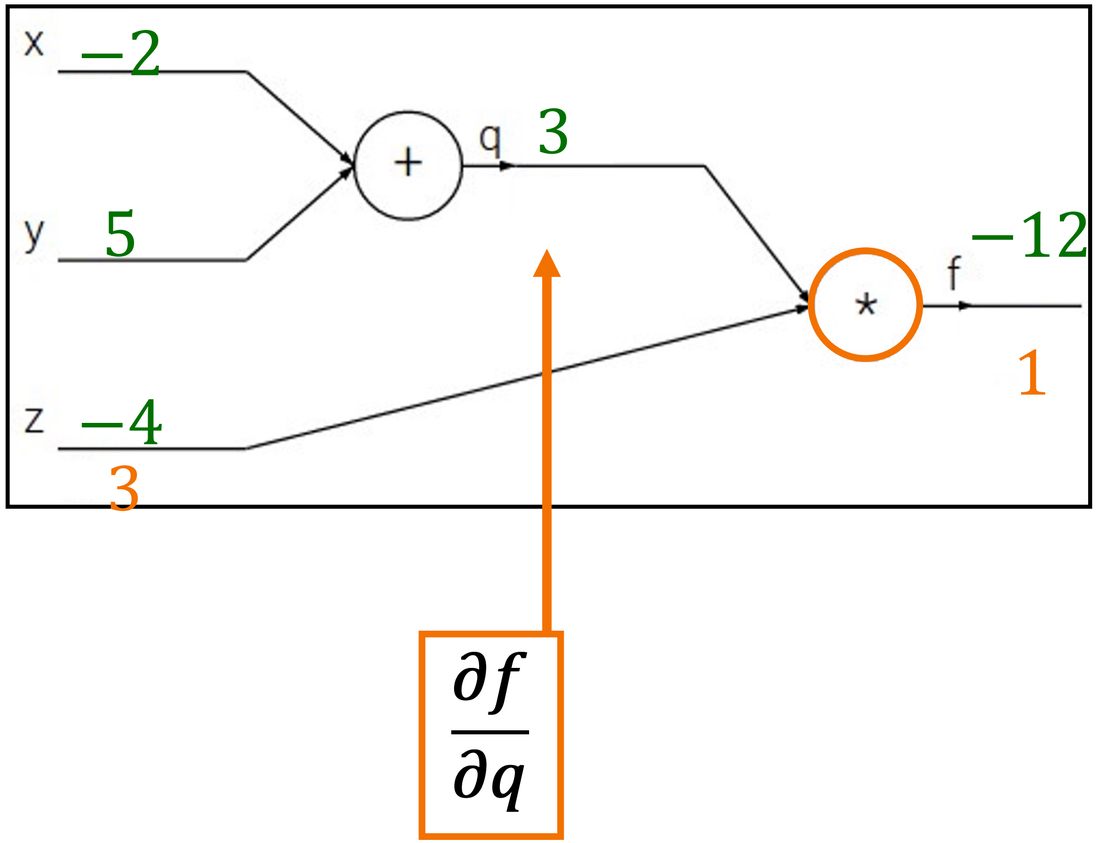
$\frac{\partial f}{\partial z}$

$= q = 3$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. Forward pass: Compute outputs

$$q = x + y \qquad f = q \cdot z$$

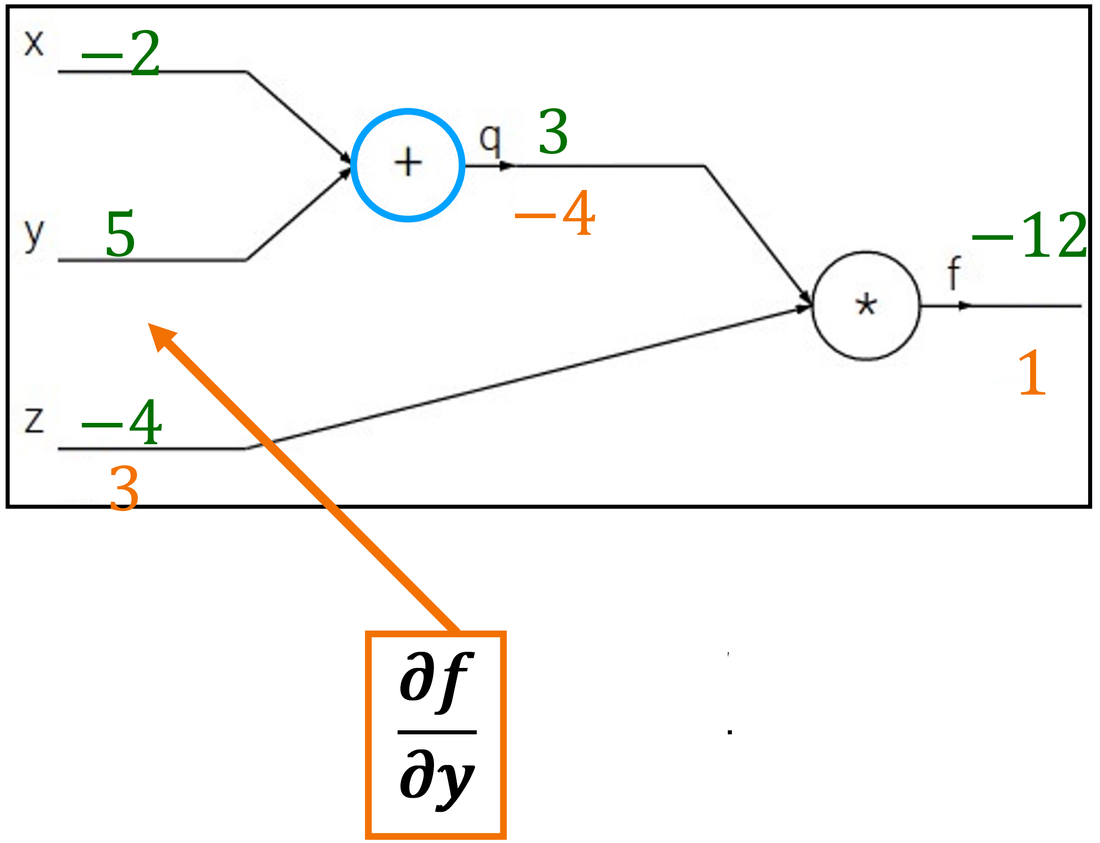
2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. Forward pass: Compute outputs

$$q = x + y \quad f = q \cdot z$$

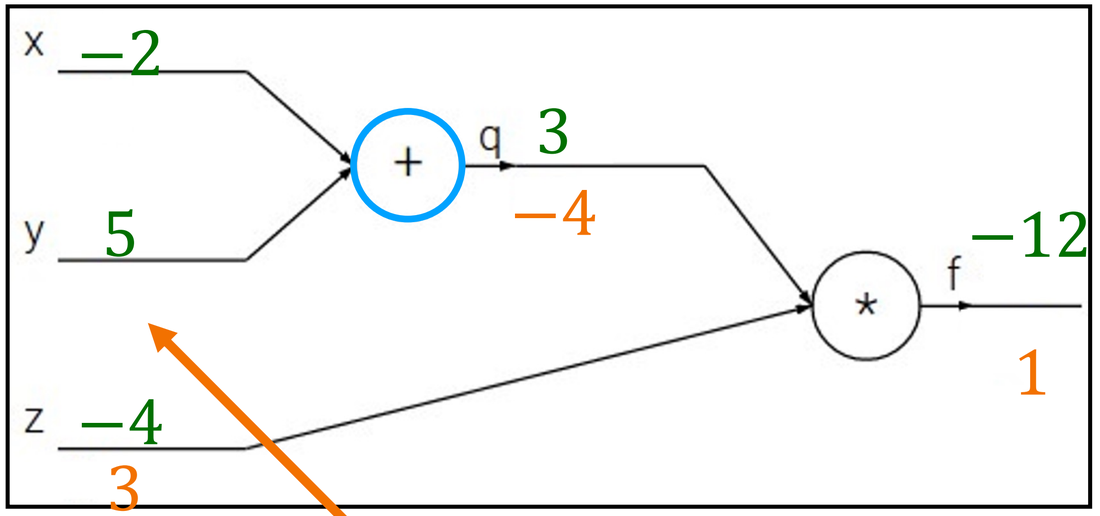
2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

e.g. $x = -2, y = 5, z = -4$



1. Forward pass: Compute outputs

$$q = x + y \quad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

"the chain rule"

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

Downstream Gradient

Local Gradient

Upstream Gradient

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

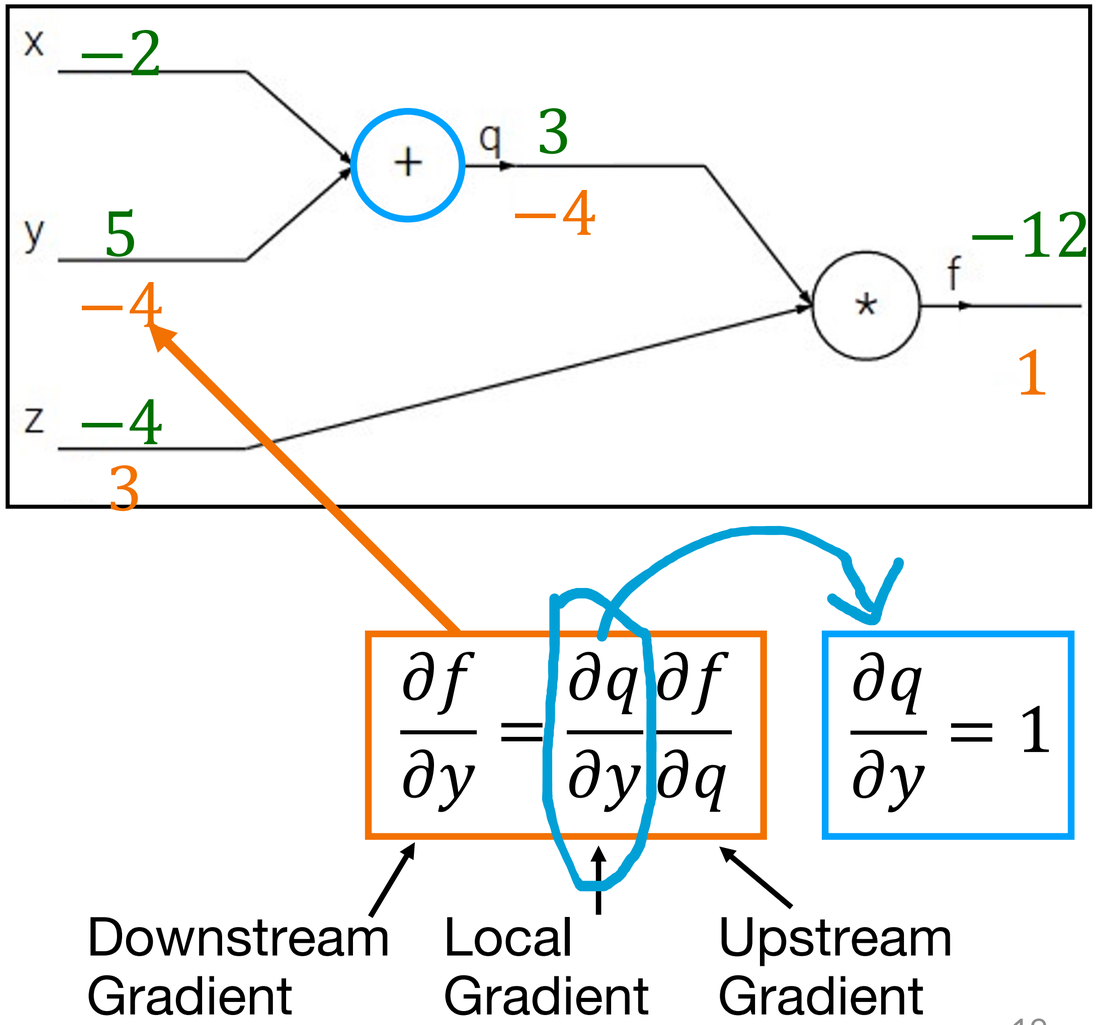
e.g. $x = -2, y = 5, z = -4$

1. Forward pass: Compute outputs

$$q = x + y \qquad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

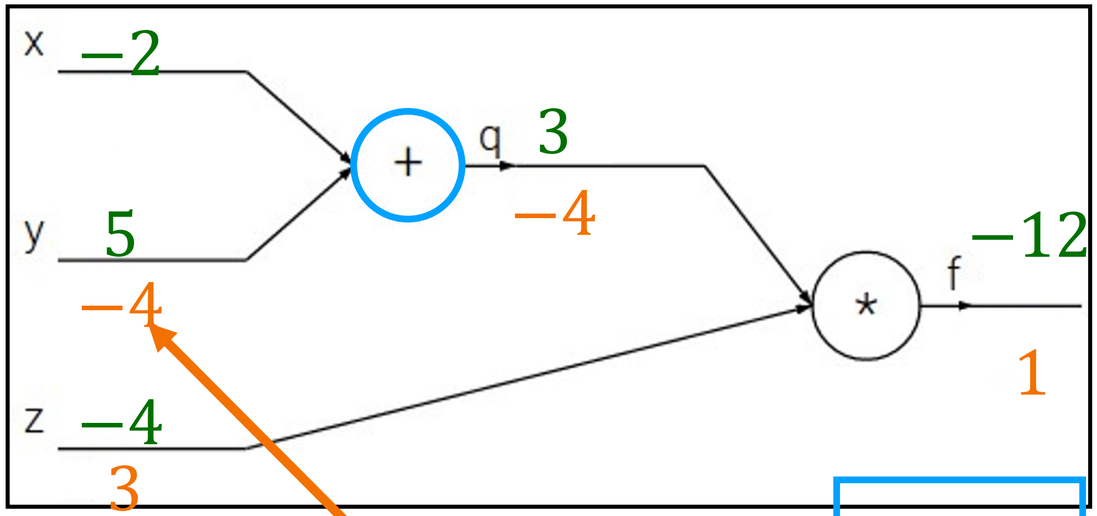
e.g. $x = -2, y = 5, z = -4$

1. Forward pass: Compute outputs

$$q = x + y \qquad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial q}{\partial y} = 1$$

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q}$$

$$\frac{\partial f}{\partial q} = z = -4$$

Downstream Gradient | Local Gradient | Upstream Gradient

Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

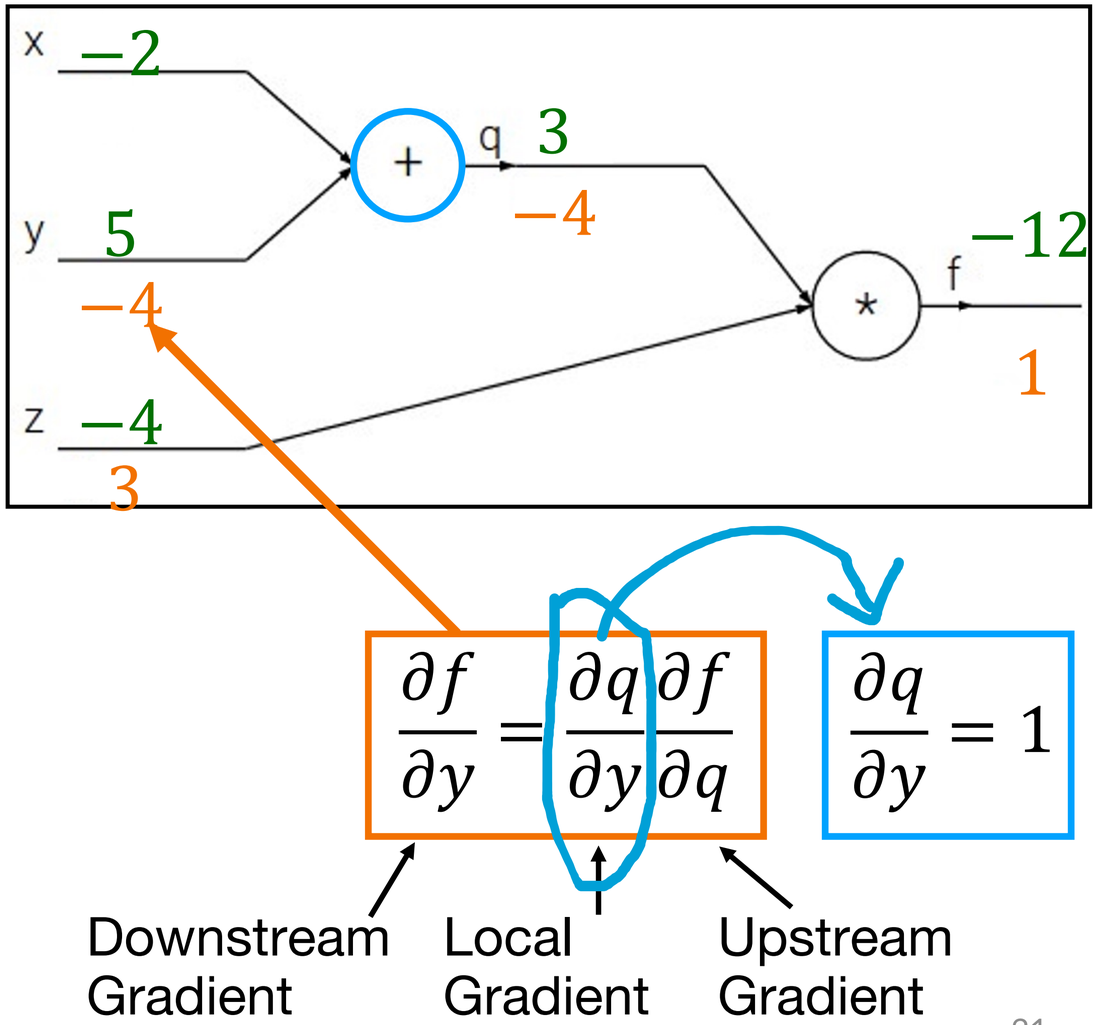
e.g. $x = -2, y = 5, z = -4$

1. Forward pass: Compute outputs

$$q = x + y \qquad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

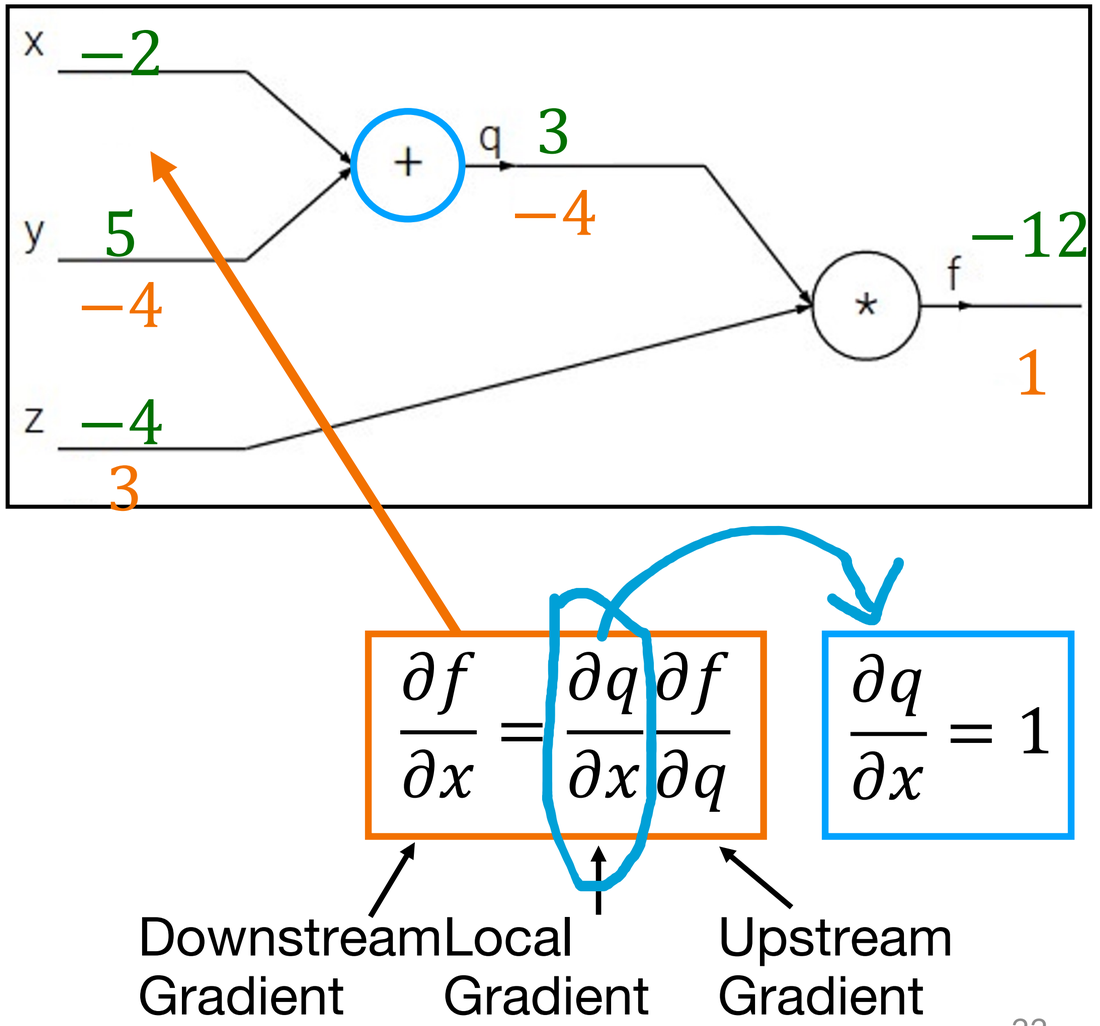
e.g. $x = -2, y = 5, z = -4$

1. Forward pass: Compute outputs

$$q = x + y \qquad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: Simple Example

$$f(x, y, z) = (x + y) \cdot z$$

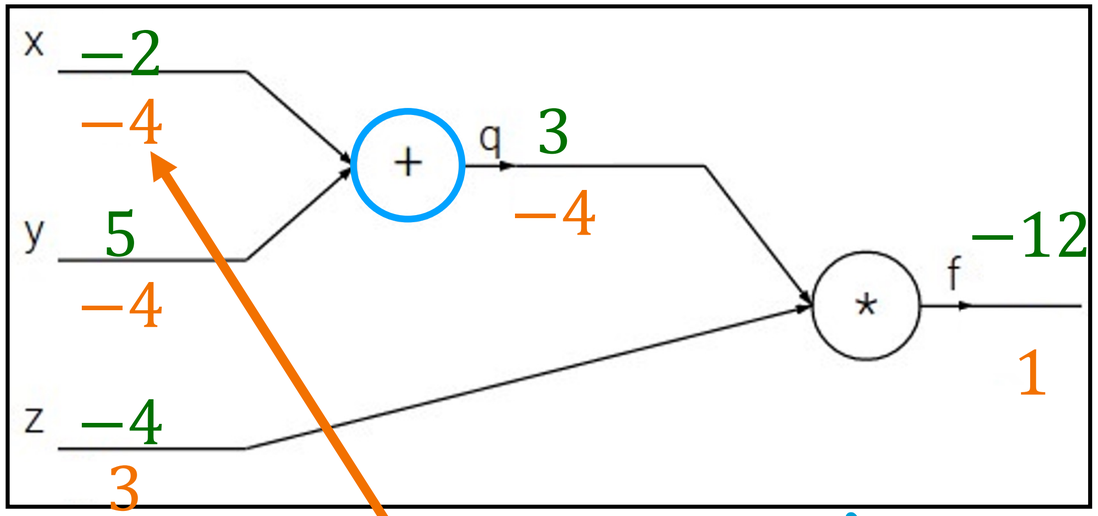
e.g. $x = -2, y = 5, z = -4$

1. Forward pass: Compute outputs

$$q = x + y \qquad f = q \cdot z$$

2. Backward pass: Compute derivatives

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



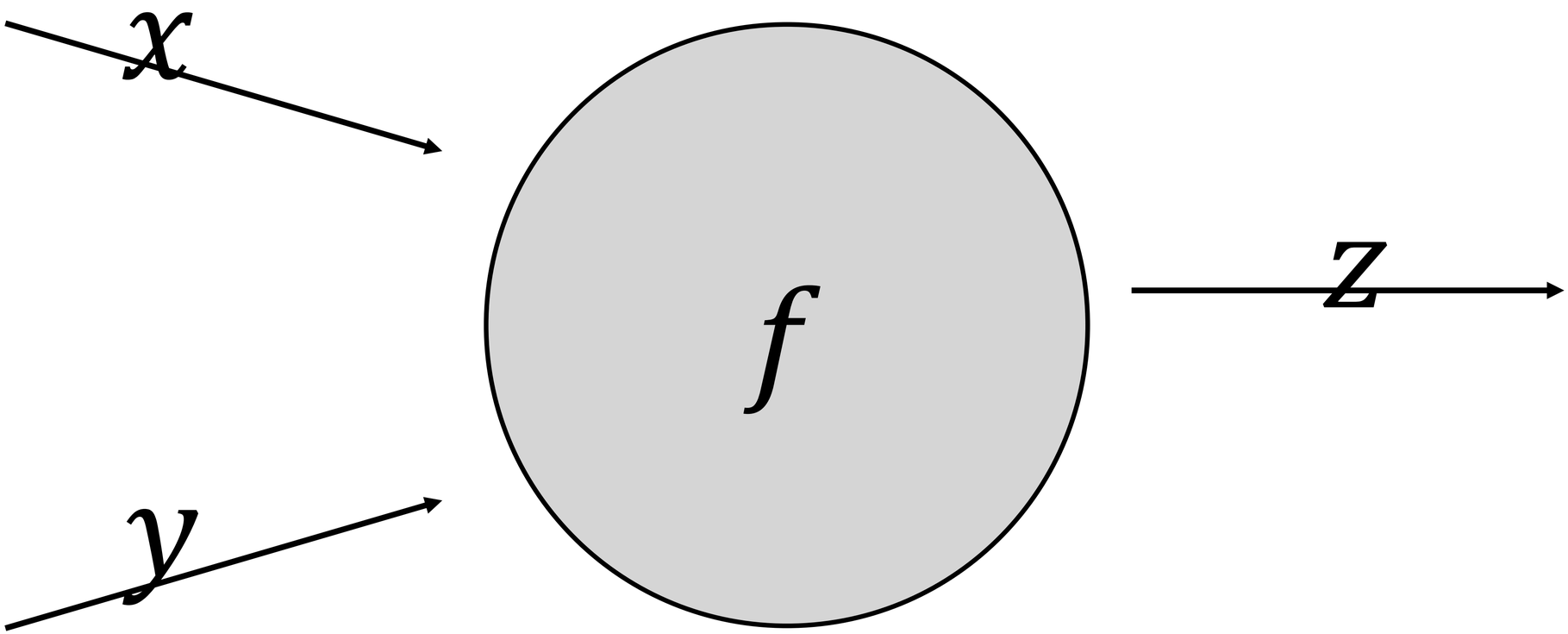
$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x}\frac{\partial f}{\partial q}$$

$$\frac{\partial q}{\partial x} = 1$$

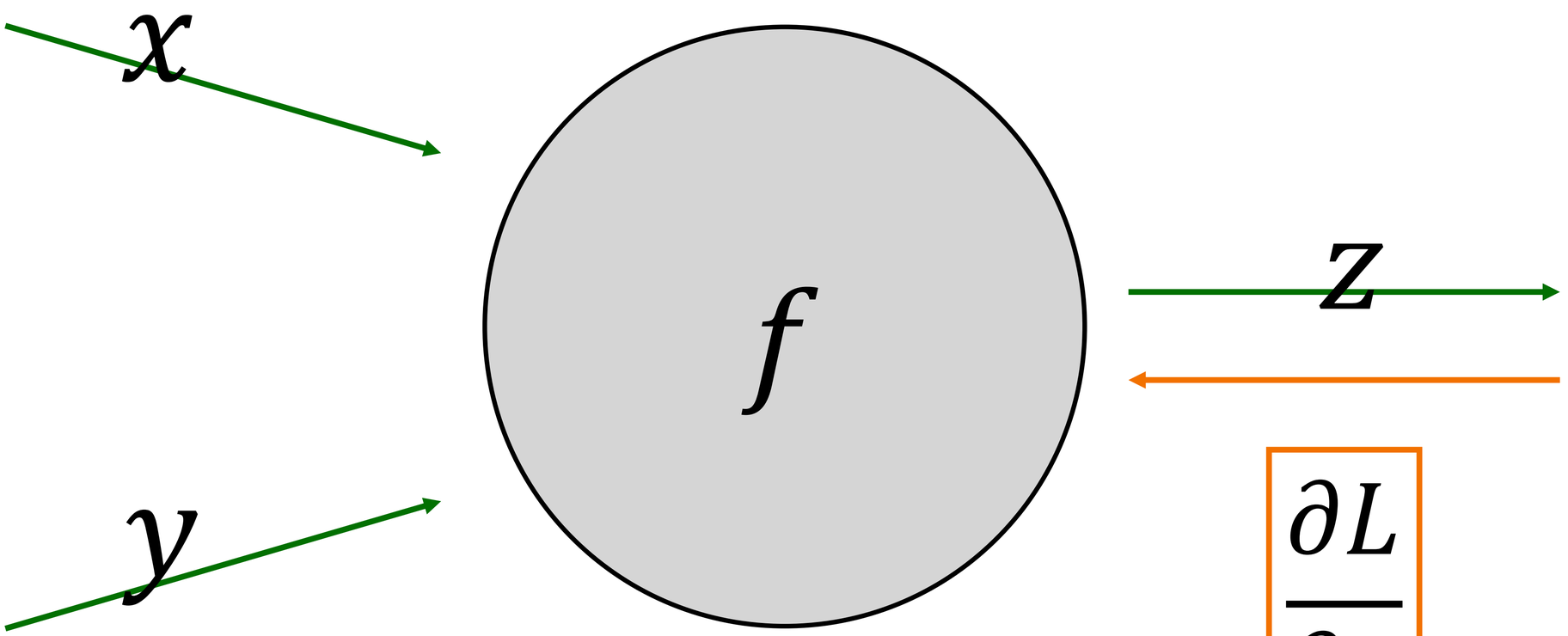
$$\frac{\partial f}{\partial q} = z = -4$$

Downstream Gradient | Local Gradient | Upstream Gradient

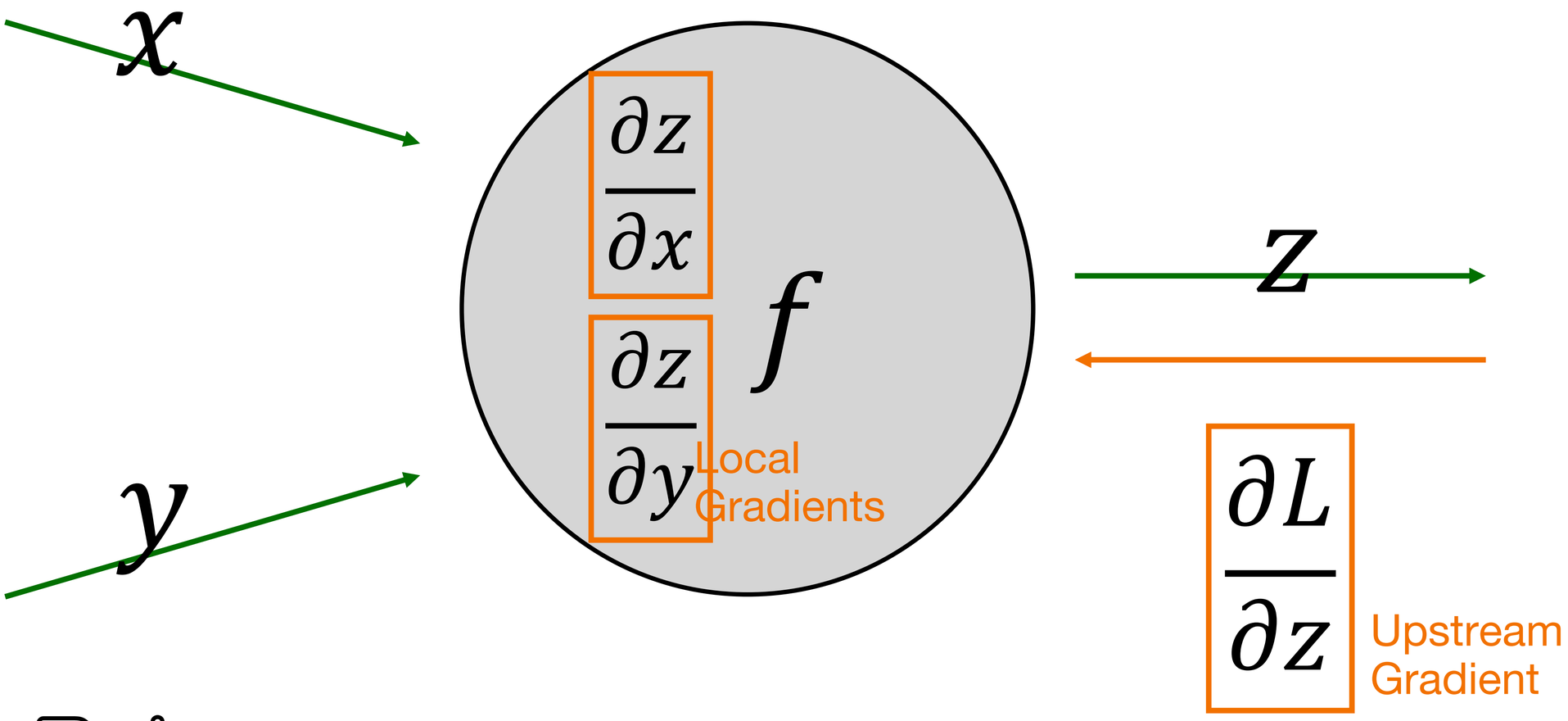
Local Properties of Backpropagation



Local Properties of Backpropagation

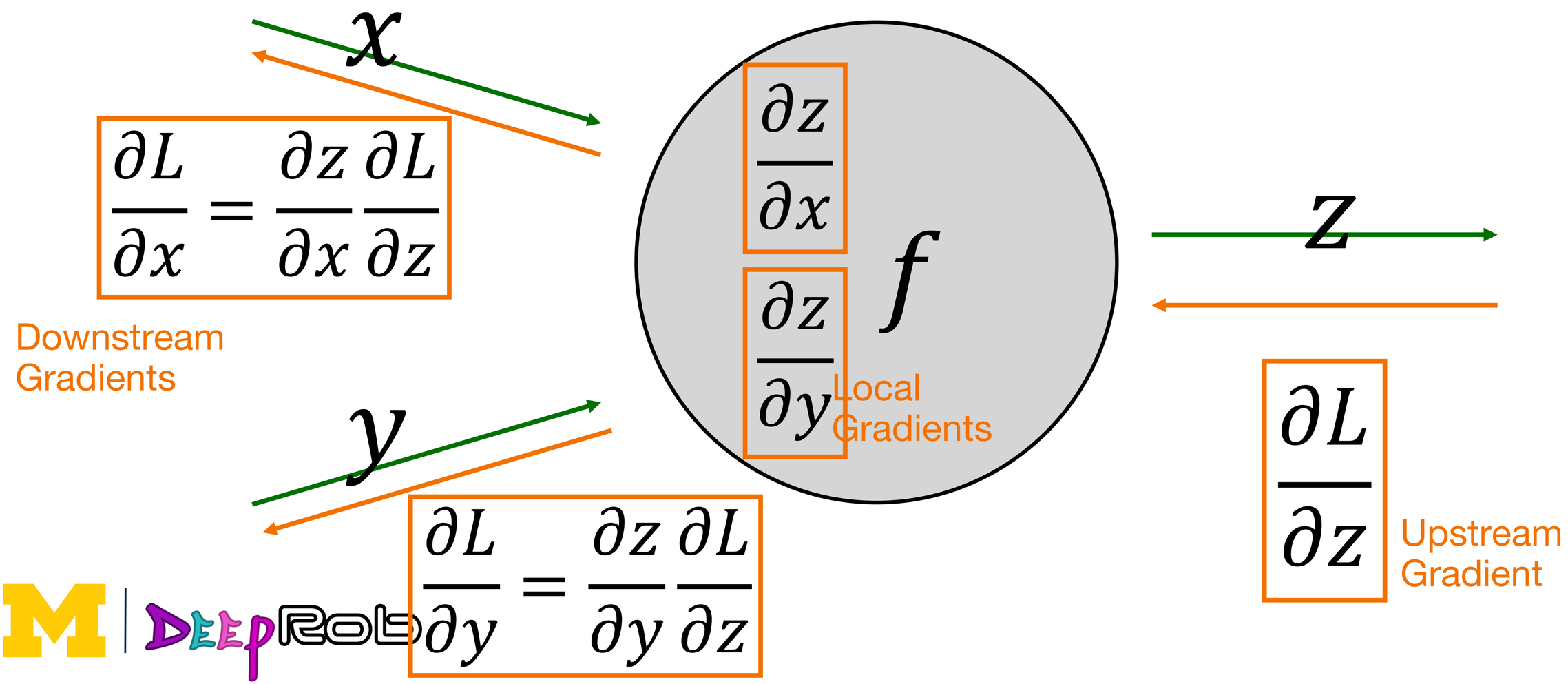


Local Properties of Backpropagation

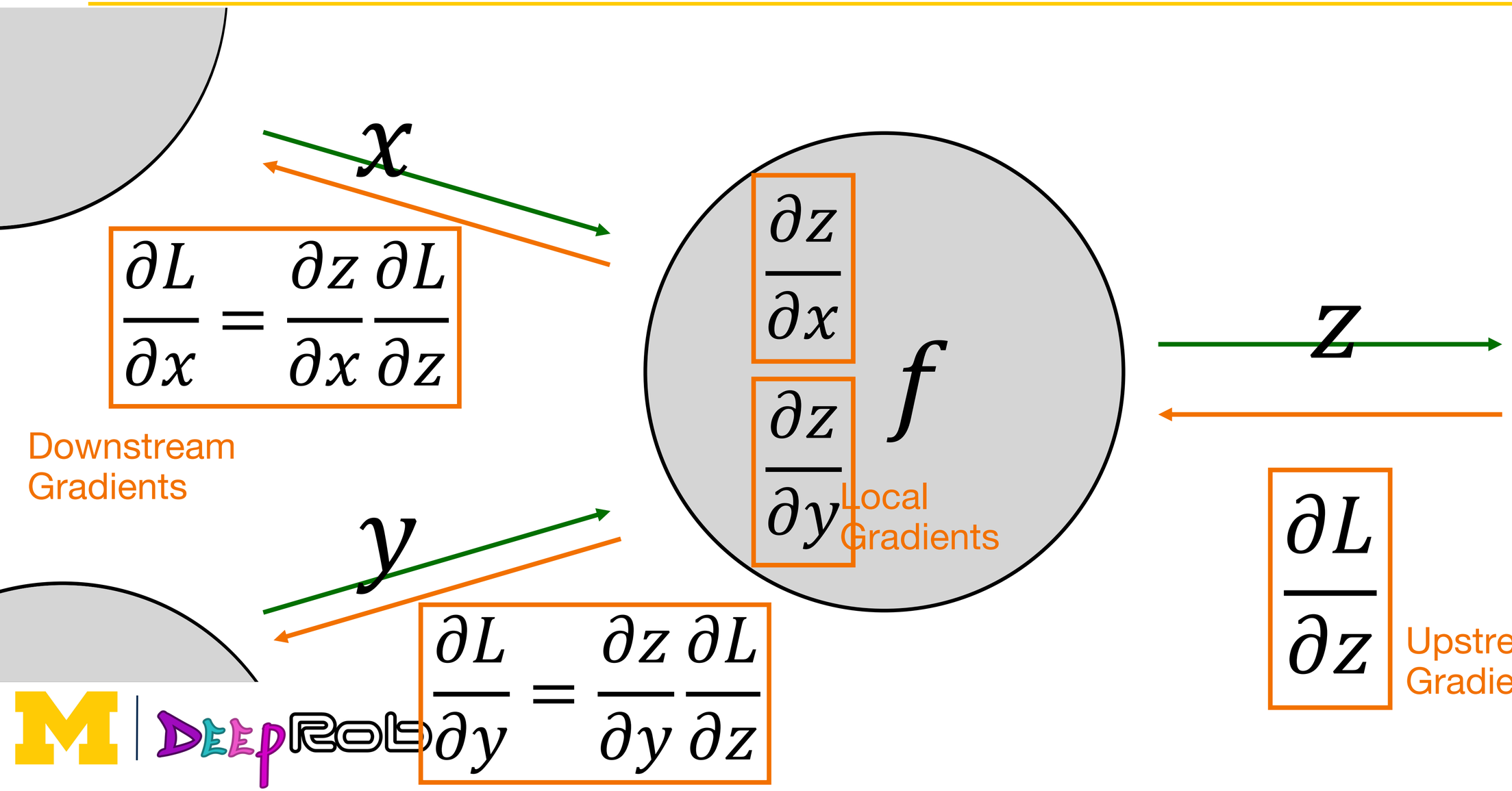


Local Properties of Backpropagation

"the chain rule"



Local Properties of Backpropagation



Another example

$$f(x, w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w2)}}$$

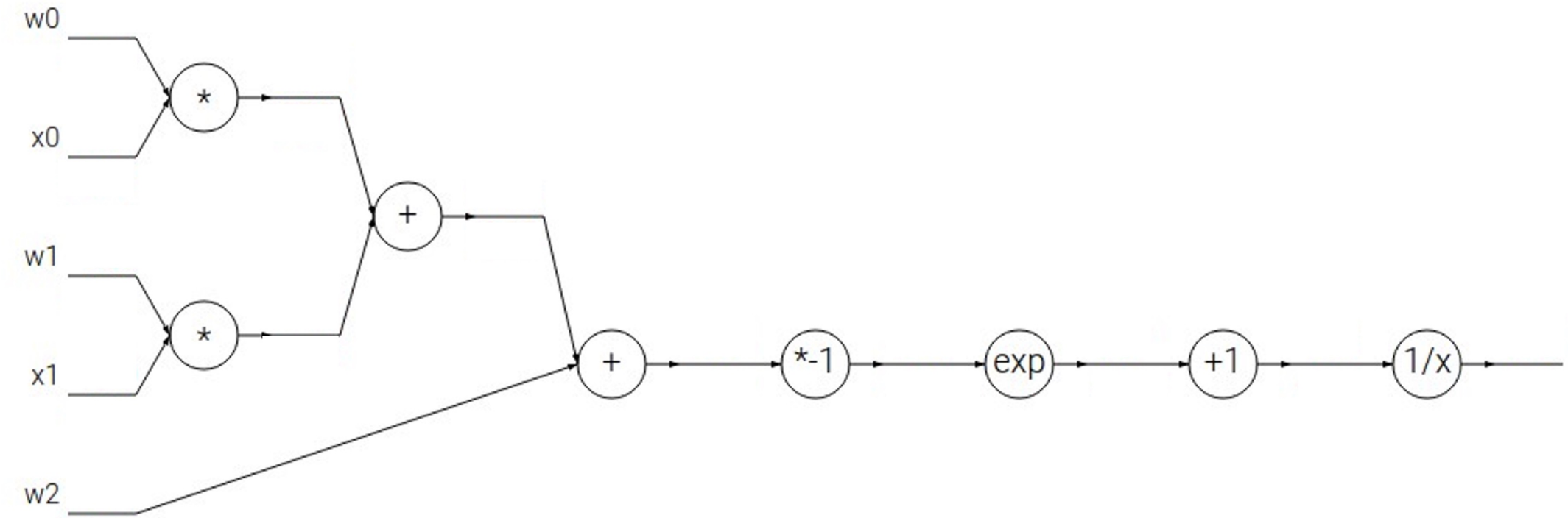
Another example

$$f(x,w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w2)}}$$

Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

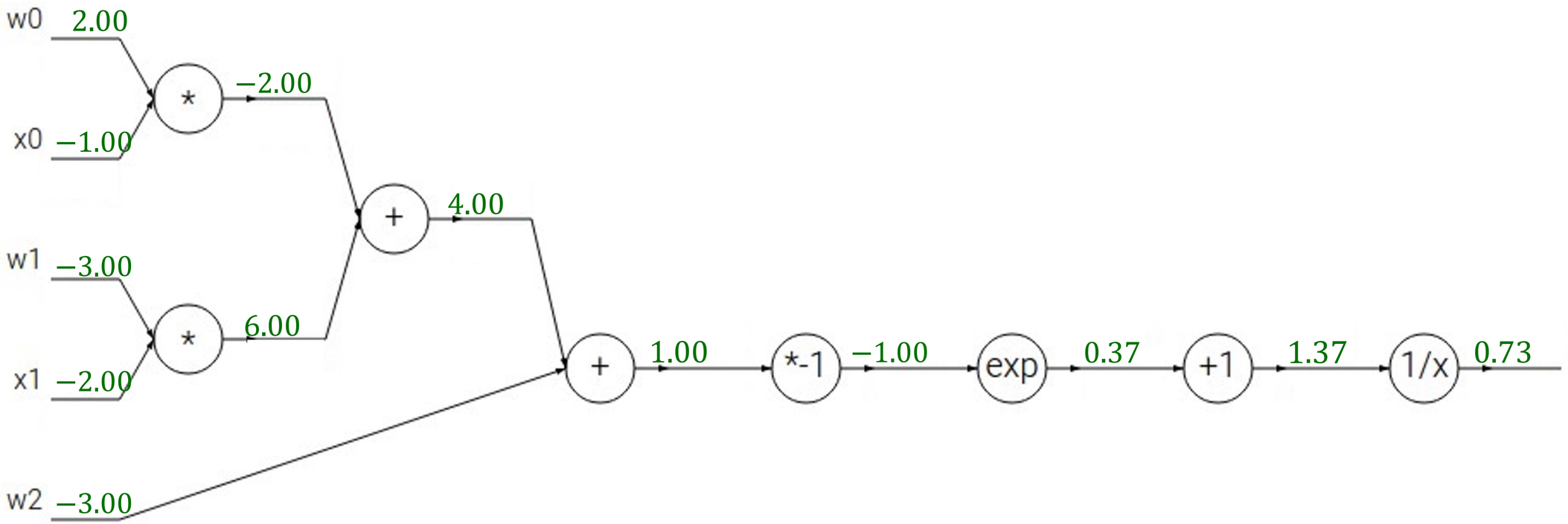
1. Forward pass: Compute outputs



Another example

$$f(x,w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w2)}}$$

1. Forward pass: Compute outputs

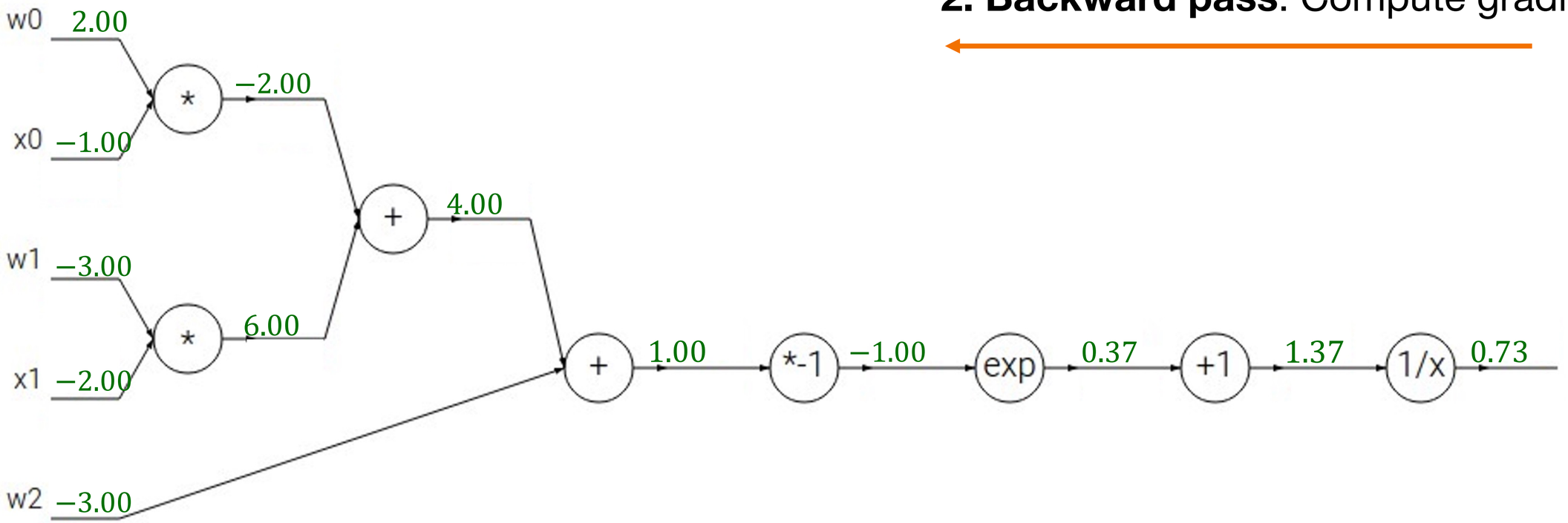


Another example

$$f(x,w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients

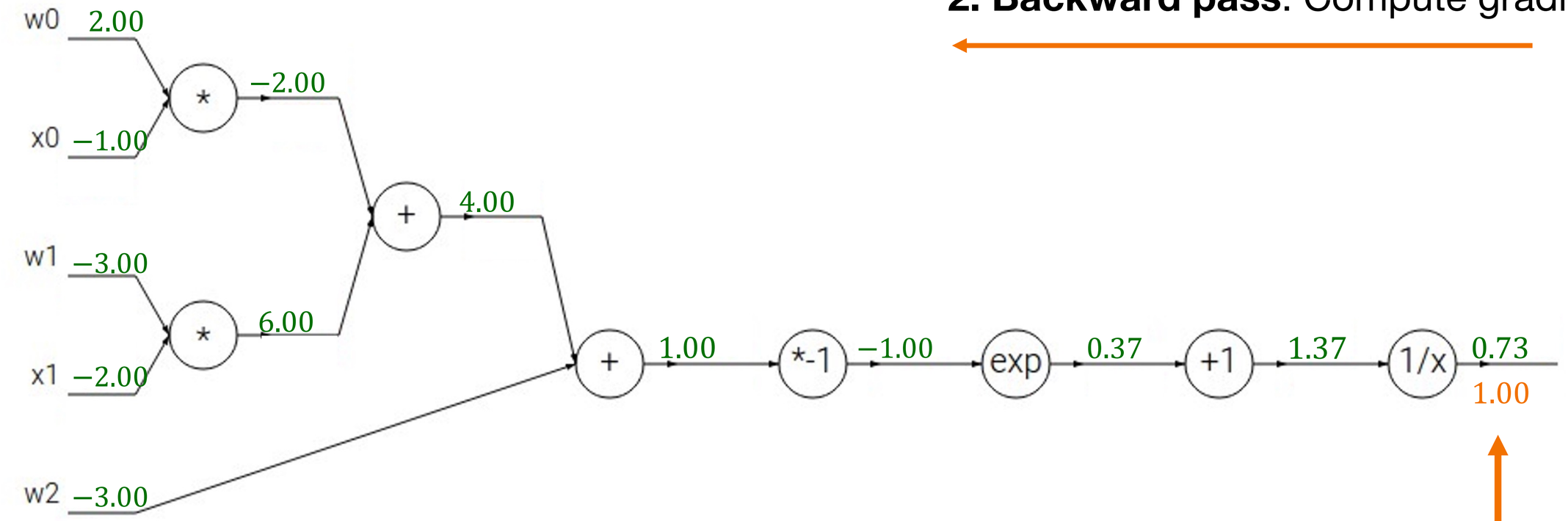


Another example

$$f(x,w) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients



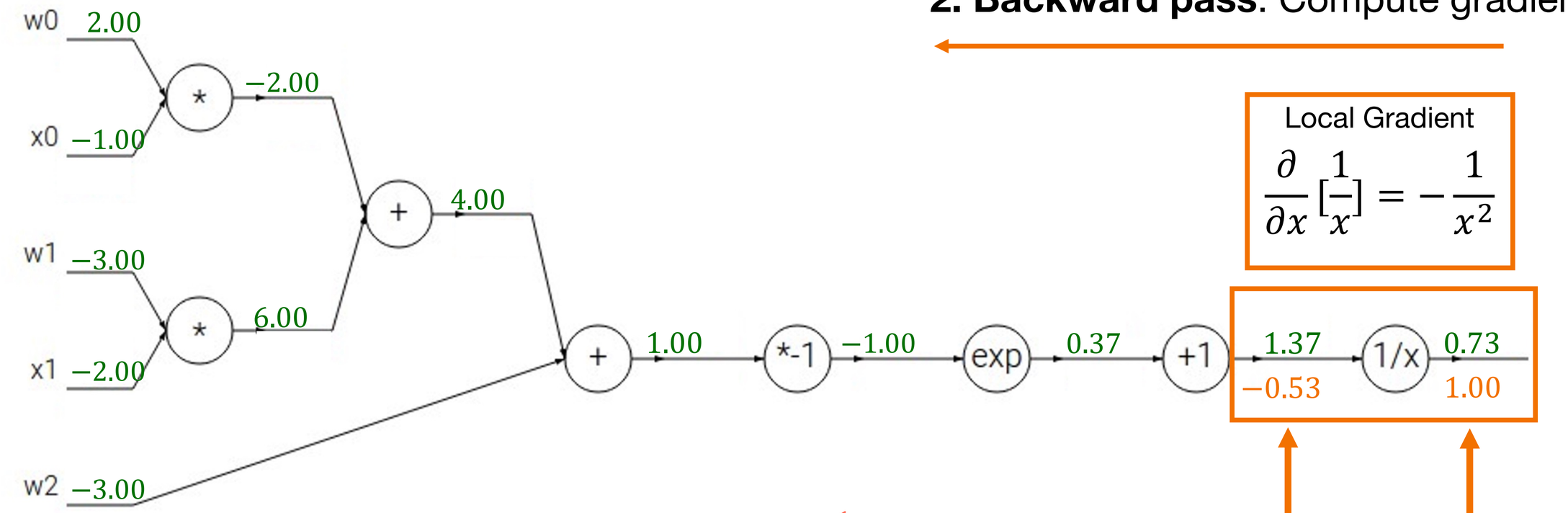
Base Case

Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients



Local Gradient

$$\frac{\partial}{\partial x}\left[\frac{1}{x}\right] = -\frac{1}{x^2}$$

$$-\frac{1}{1.37^2} = -0.53$$

Downstream Gradient

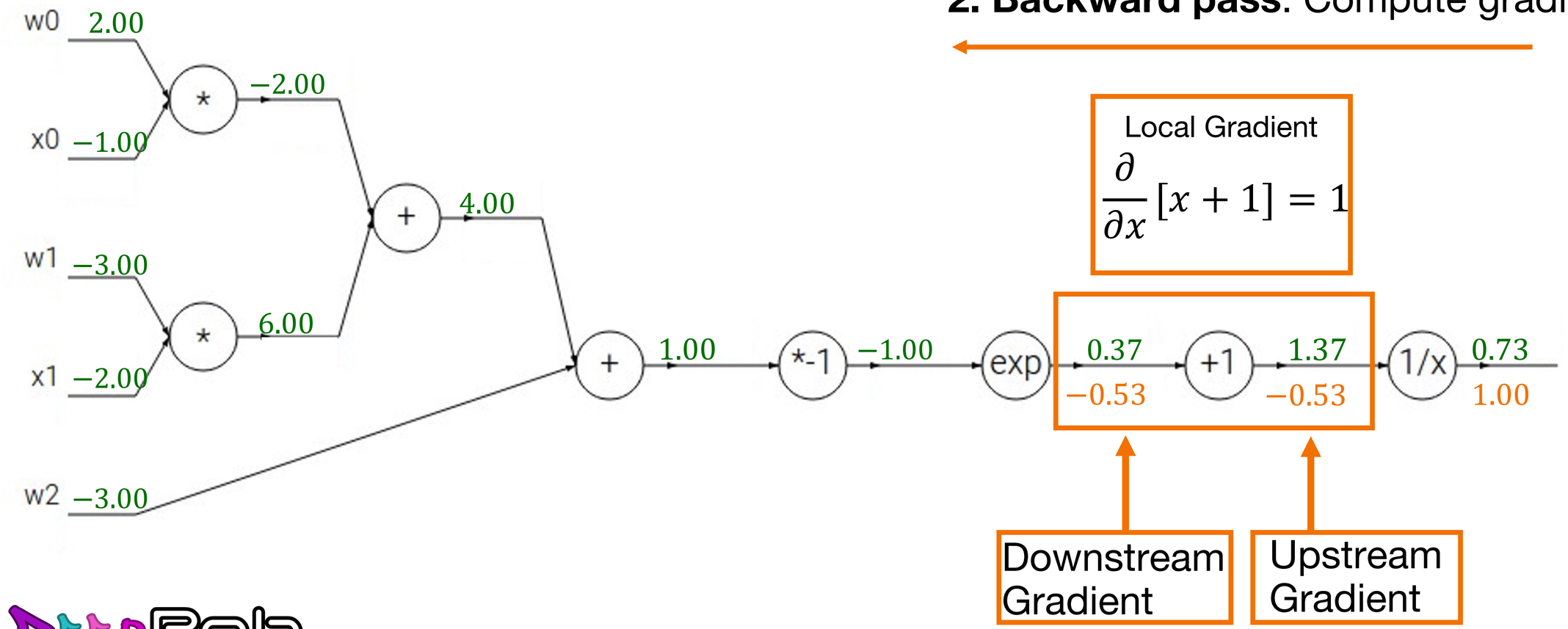
Upstream Gradient

Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients

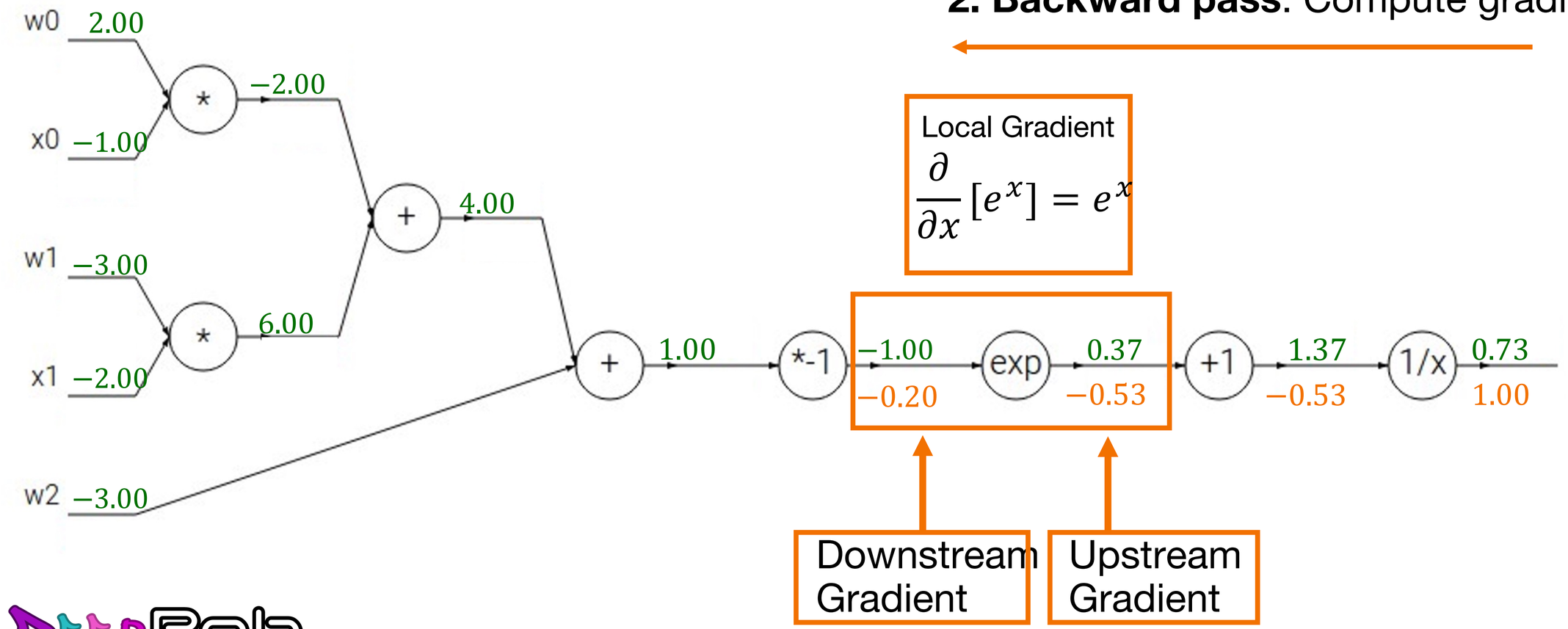


Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients

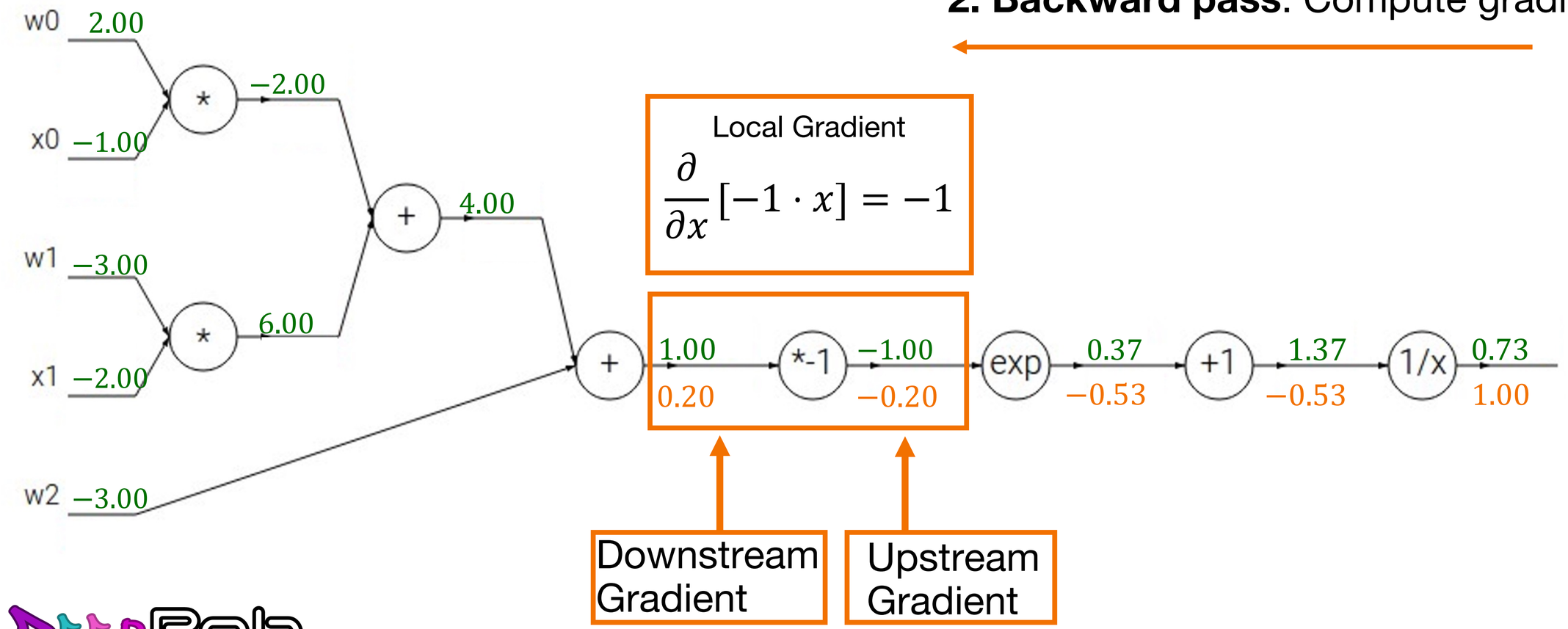


Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients

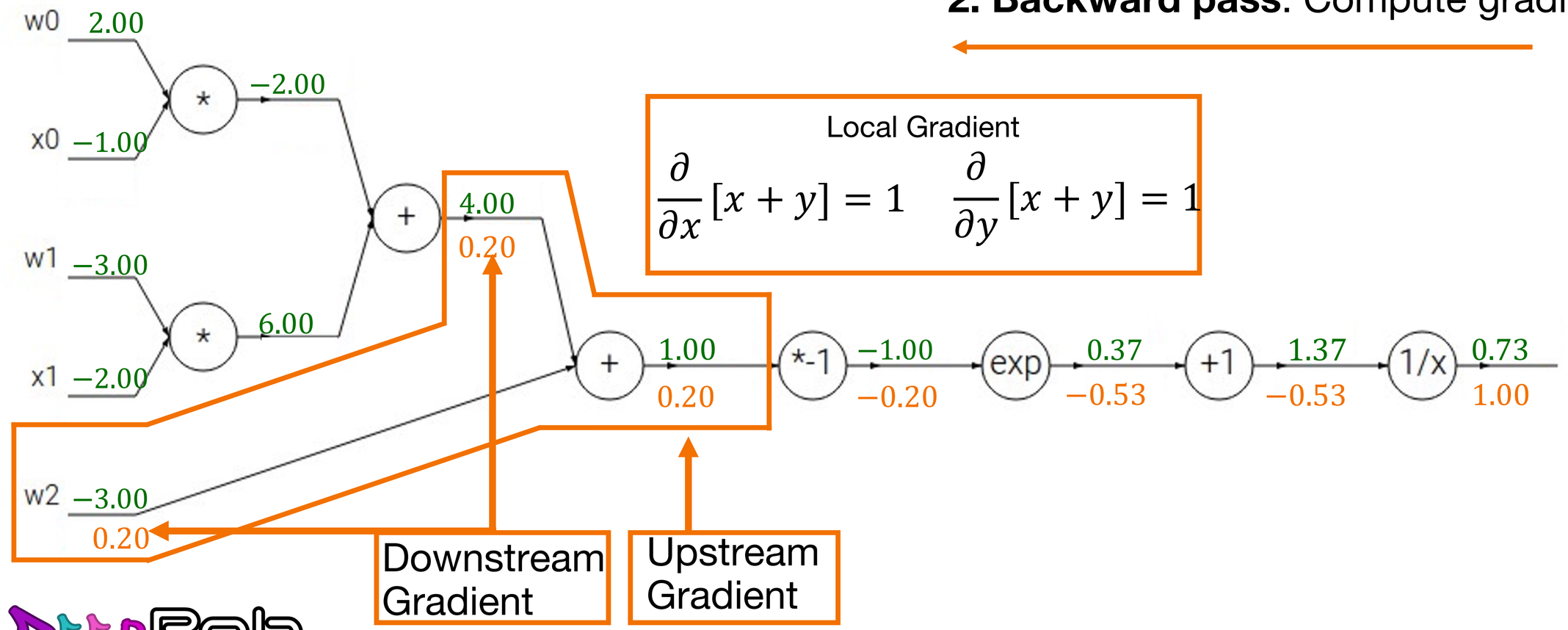


Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients

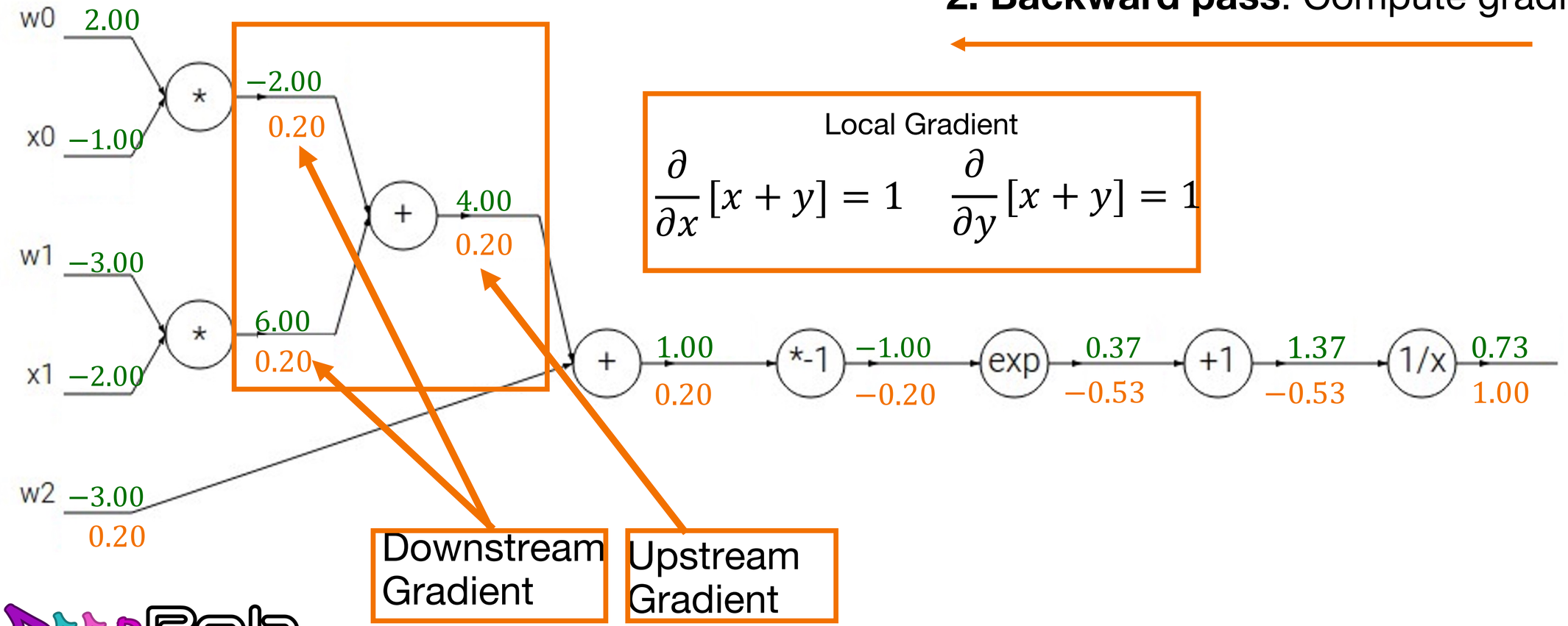


Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients

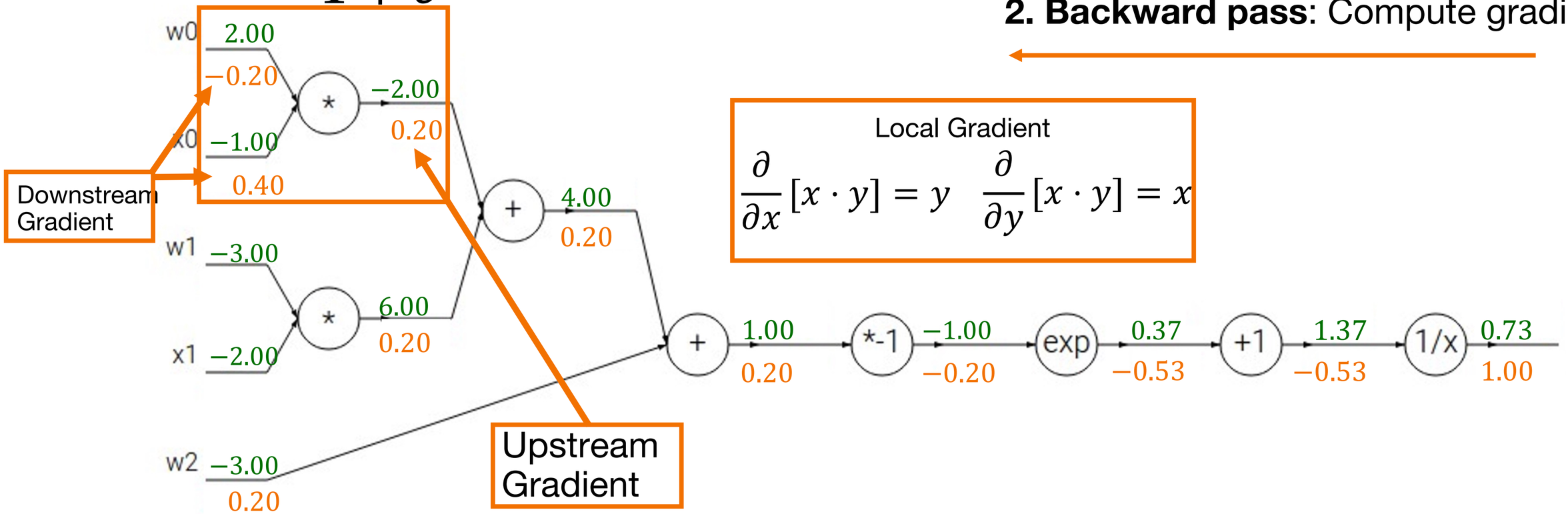


Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients



Local Gradient

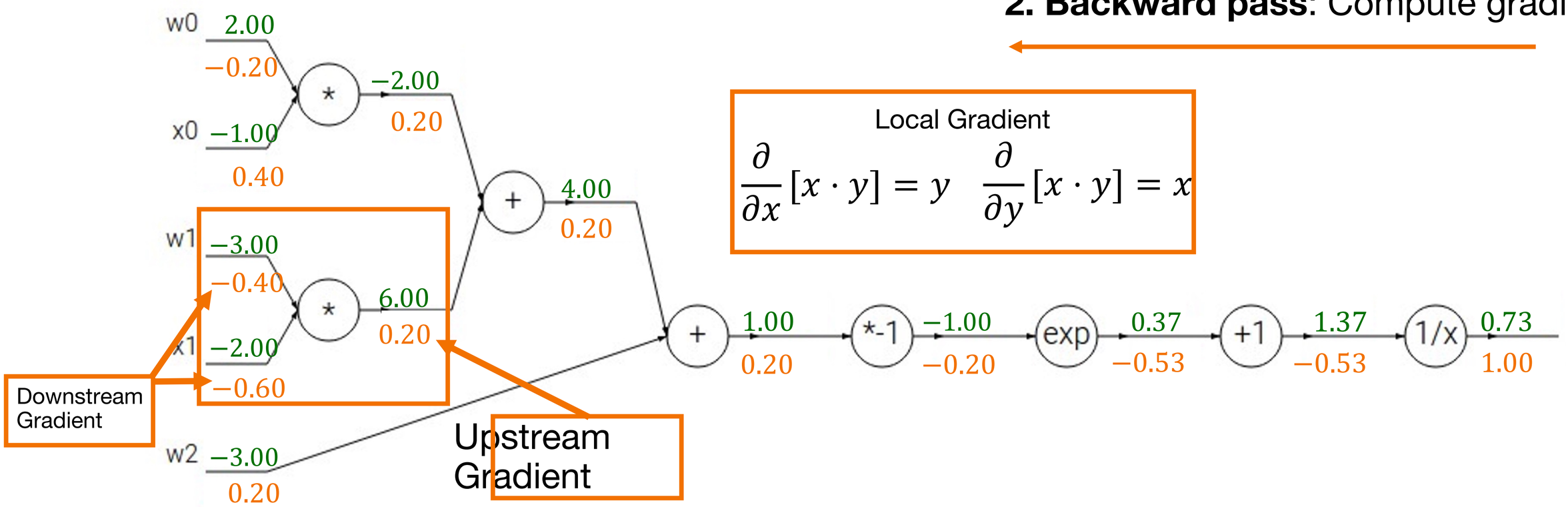
$$\frac{\partial}{\partial x}[x \cdot y] = y \quad \frac{\partial}{\partial y}[x \cdot y] = x$$

Another example

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients



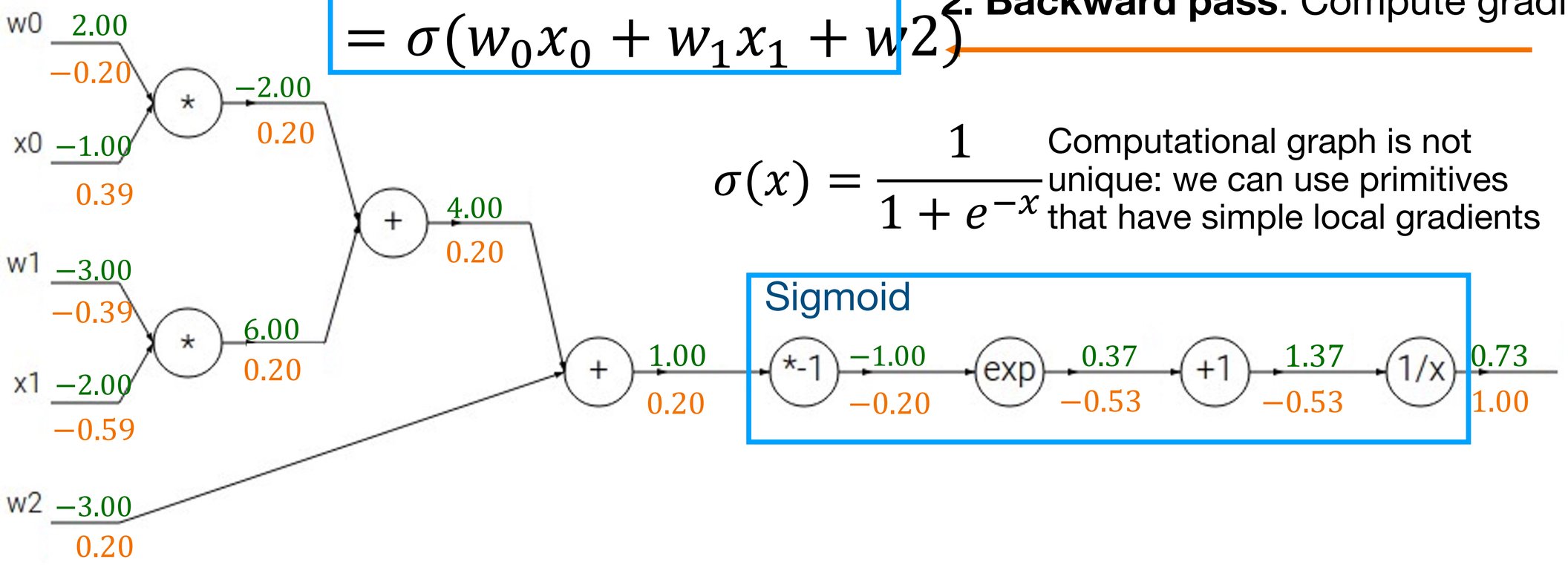
The Sigmoid

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

$$= \sigma(w_0x_0 + w_1x_1 + w2)$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients



$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Computational graph is not unique: we can use primitives that have simple local gradients

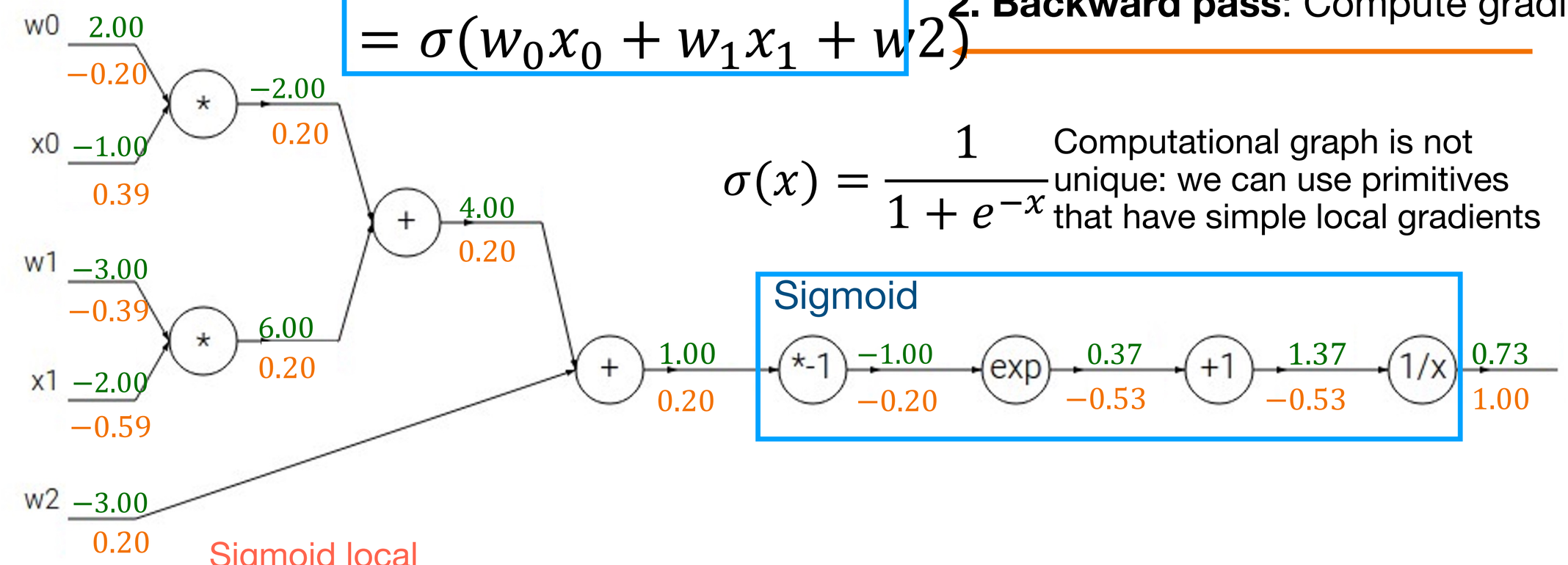
The Sigmoid

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

$$= \sigma(w_0x_0 + w_1x_1 + w2)$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients



$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Computational graph is not unique: we can use primitives that have simple local gradients

Sigmoid local gradient:

$$\frac{\partial}{\partial x}[\sigma(x)] = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

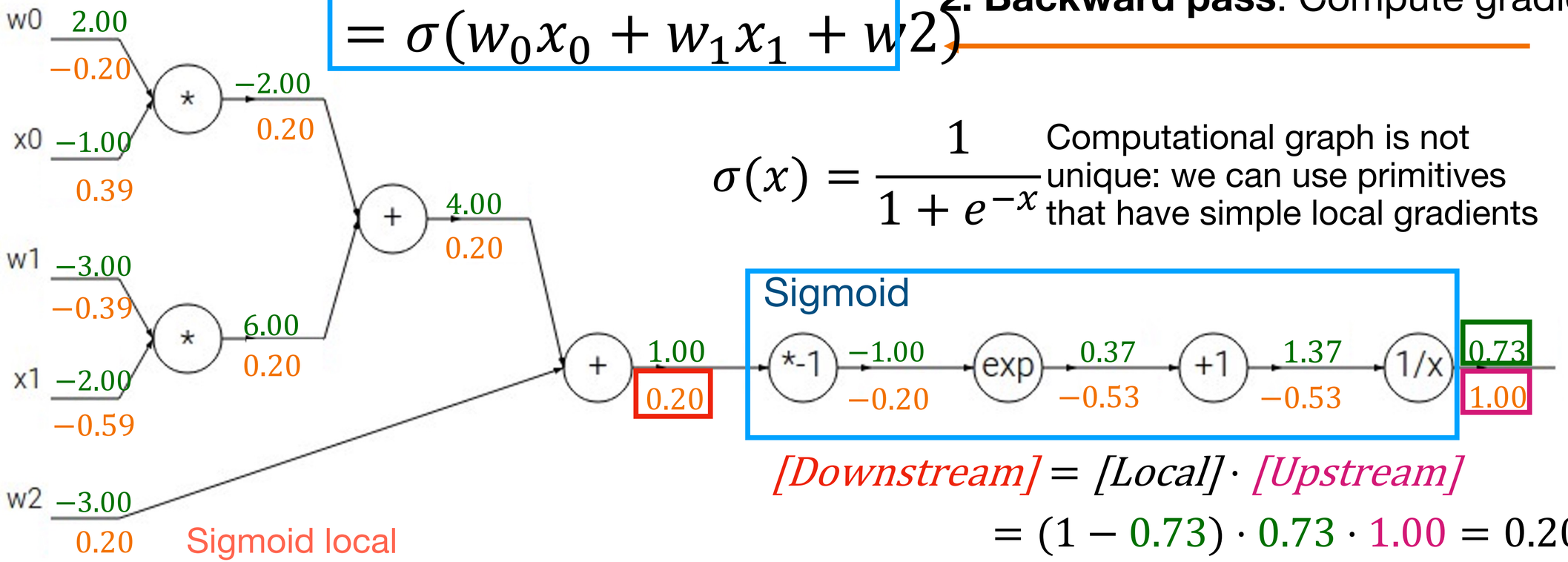
The Sigmoid

$$f(x,w) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w2)}}$$

$$= \sigma(w_0x_0 + w_1x_1 + w2)$$

1. Forward pass: Compute outputs

2. Backward pass: Compute gradients



$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Computational graph is not unique: we can use primitives that have simple local gradients

$$[Downstream] = [Local] \cdot [Upstream]$$

$$= (1 - 0.73) \cdot 0.73 \cdot 1.00 = 0.20$$

Sigmoid local gradient:

$$\frac{\partial}{\partial x}[\sigma(x)] = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

Other Common Functions

$$f(x) = e^x \longrightarrow \frac{df}{dx} = e^x$$

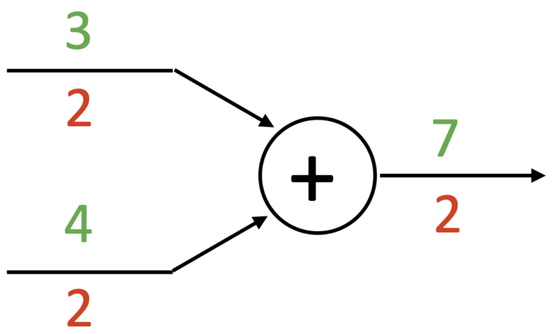
$$f_a(x) = ax \longrightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \longrightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \longrightarrow \frac{df}{dx} = 1$$

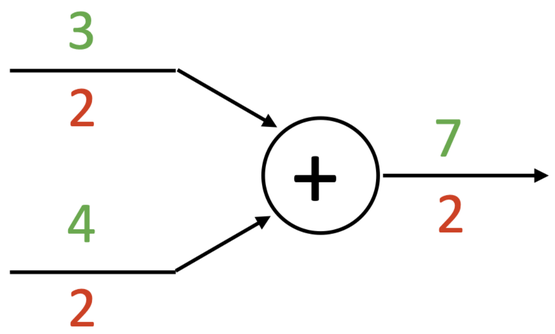
Patterns in Gradient Flow

add gate: gradient distributor

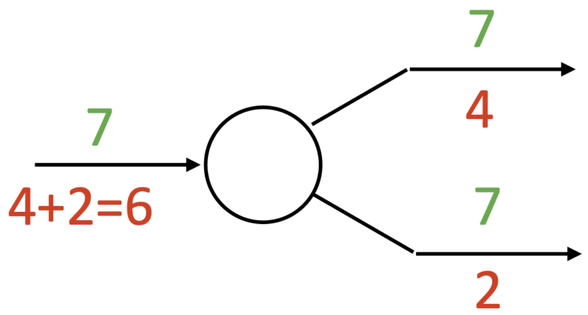


Patterns in Gradient Flow

add gate: gradient distributor

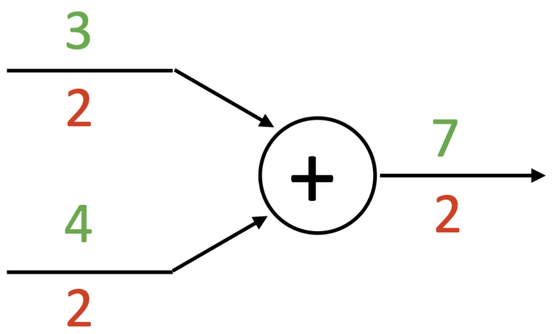


copy gate: gradient adder

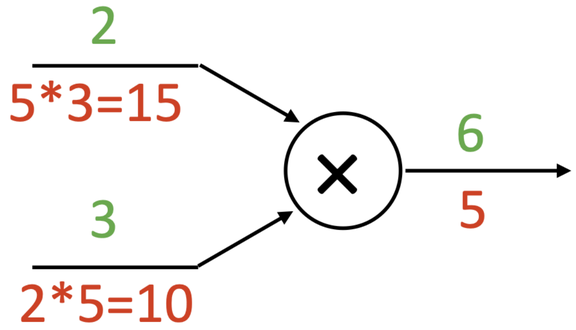


Patterns in Gradient Flow

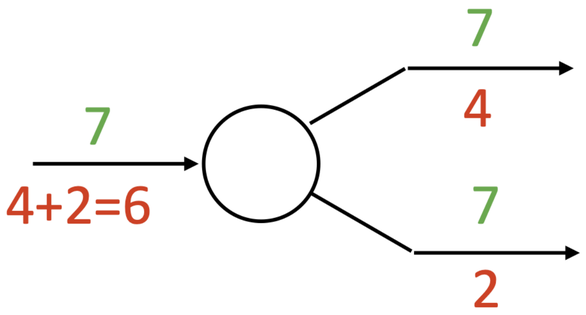
add gate: gradient distributor



mul gate: “swap multiplier”

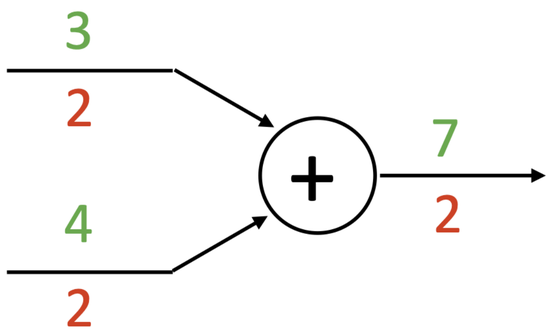


copy gate: gradient adder

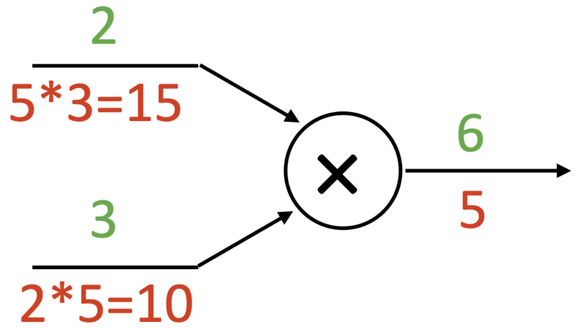


Patterns in Gradient Flow

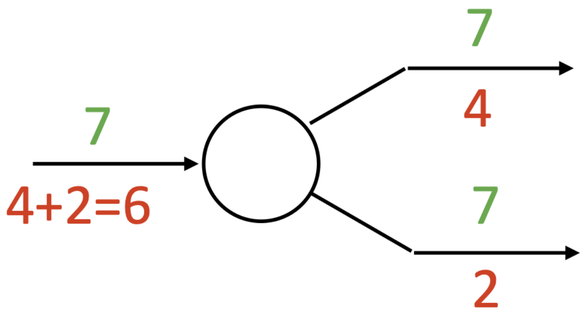
add gate: gradient distributor



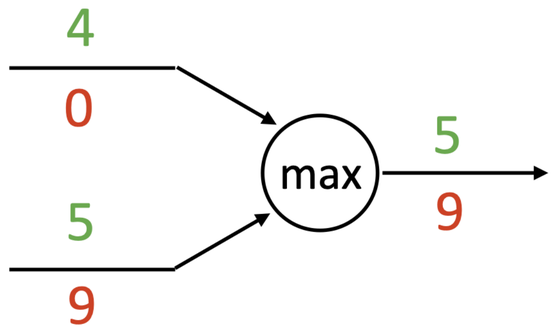
mul gate: “swap multiplier”



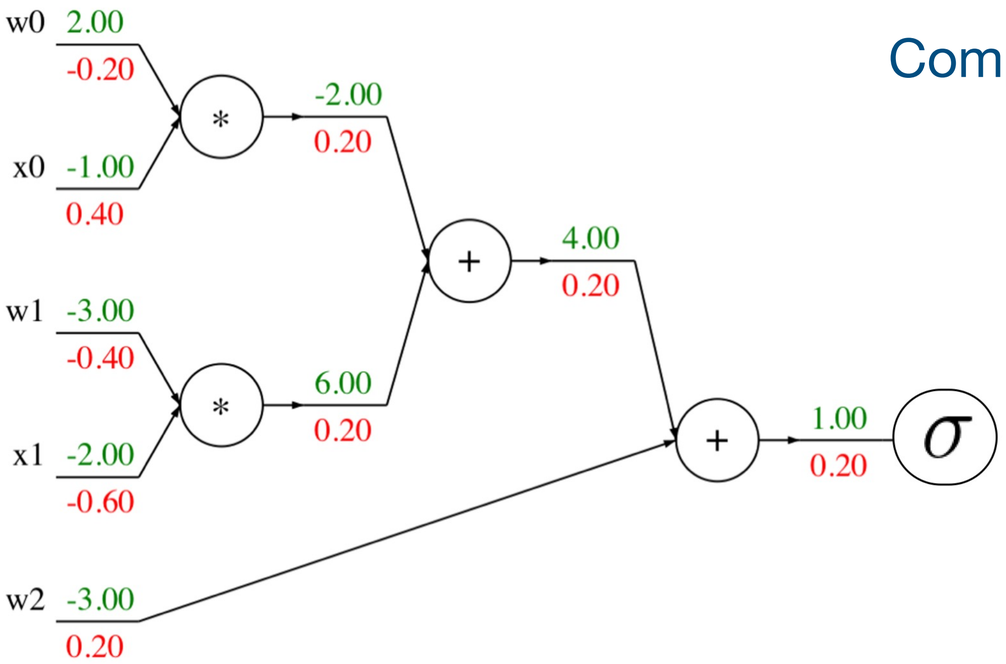
copy gate: gradient adder



max gate: gradient router



Backprop Implementation: “Flat” gradient code

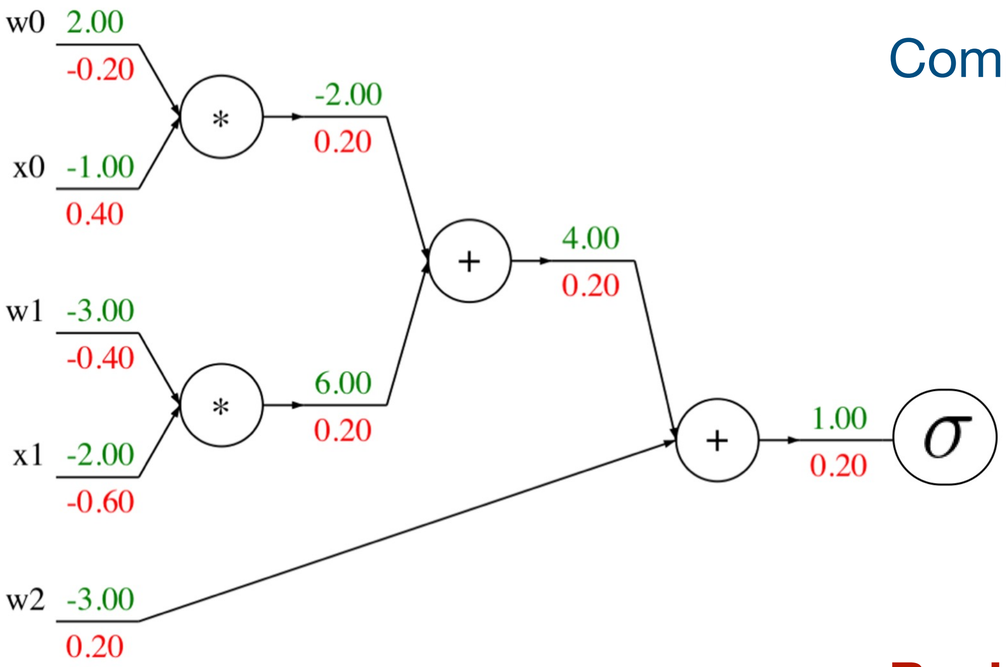


Forward pass:

Compute outputs

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

Backprop Implementation: “Flat” gradient code



Forward pass:

Compute outputs

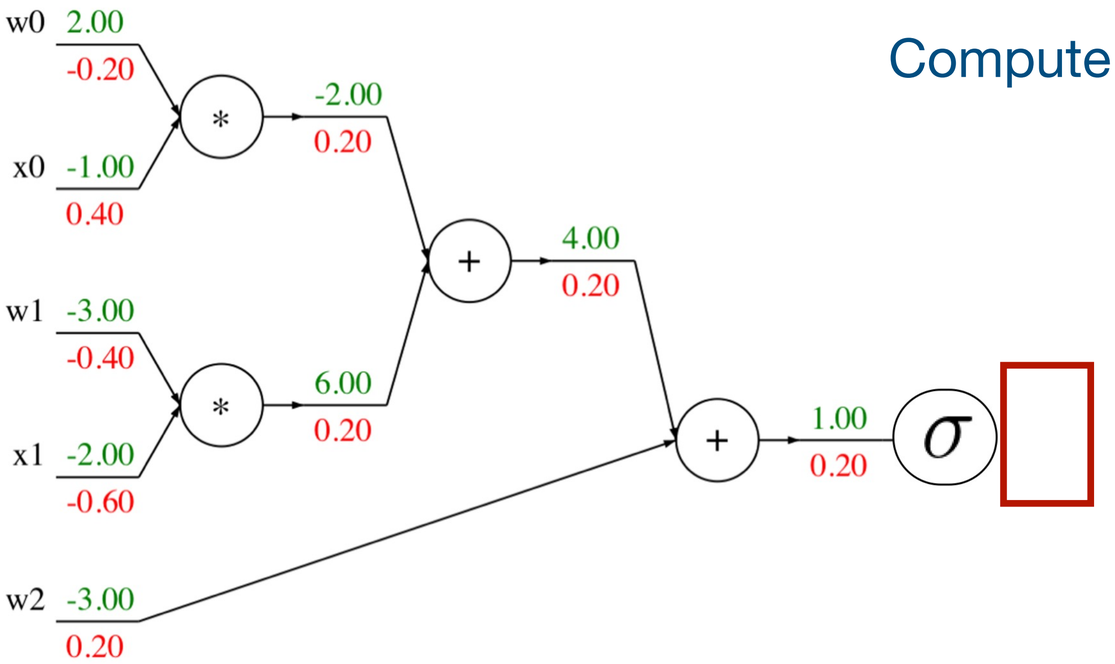
```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

Backward pass:

Compute gradients

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” gradient code



Forward pass:

Compute outputs

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

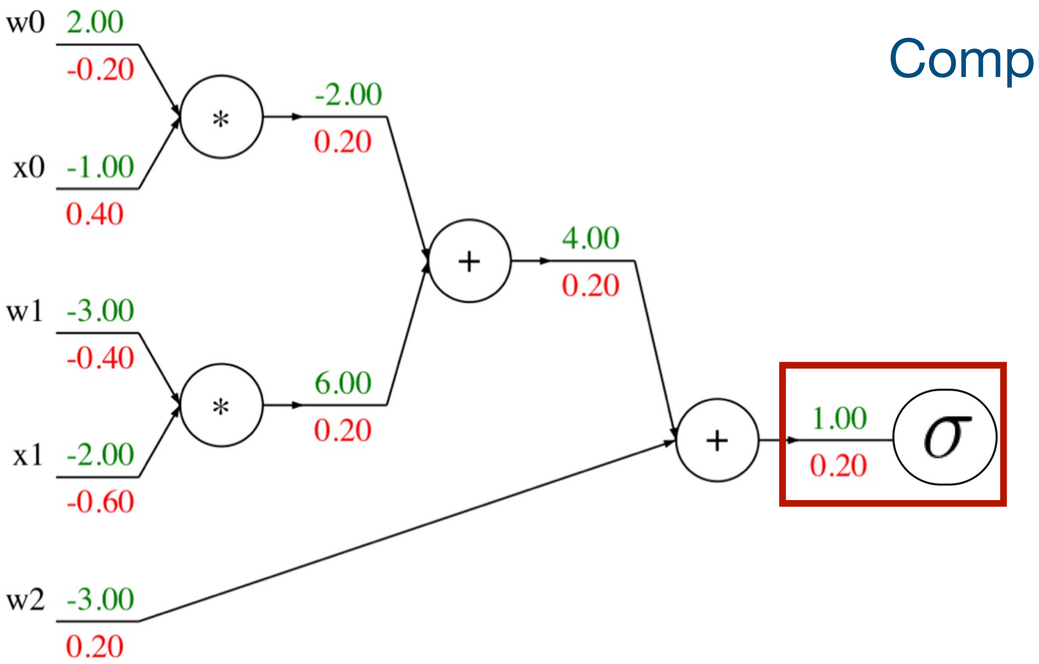
Base case

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Backward pass:

Compute gradients

Backprop Implementation: “Flat” gradient code



Forward pass:

Compute outputs

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

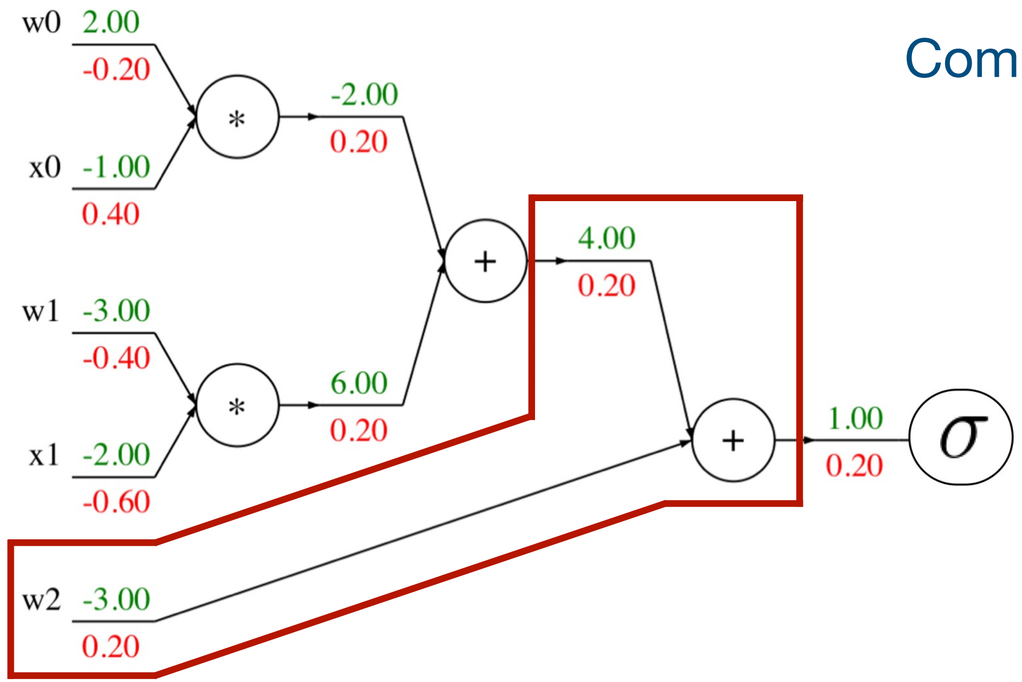
Sigmoid

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Backward pass:

Compute gradients

Backprop Implementation: “Flat” gradient code



Forward pass:

Compute outputs

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

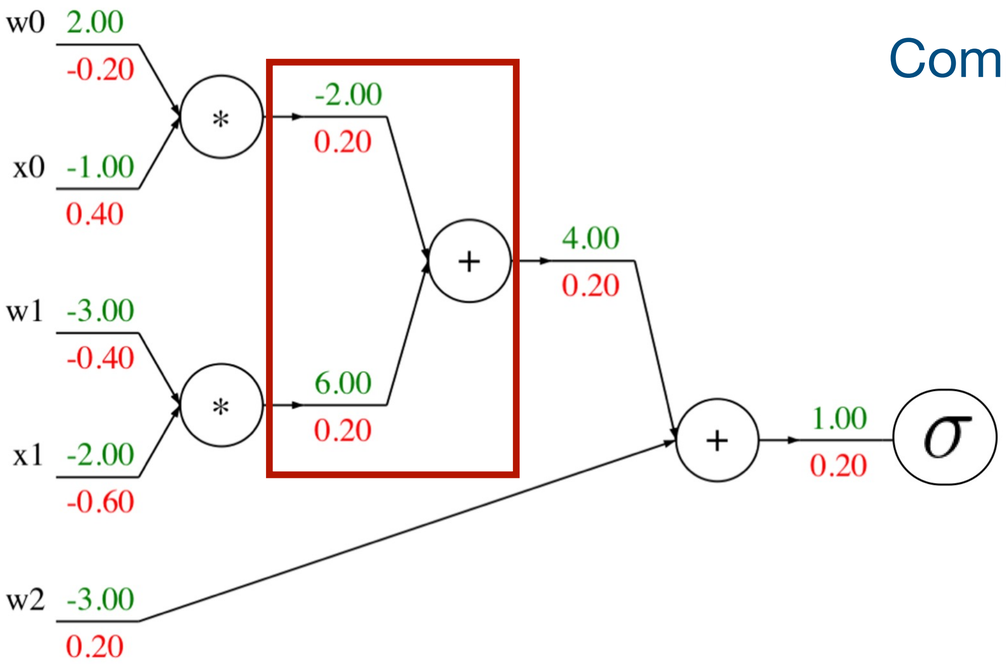
Add

Backward pass:

Compute gradients

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” gradient code



Forward pass:

Compute outputs

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

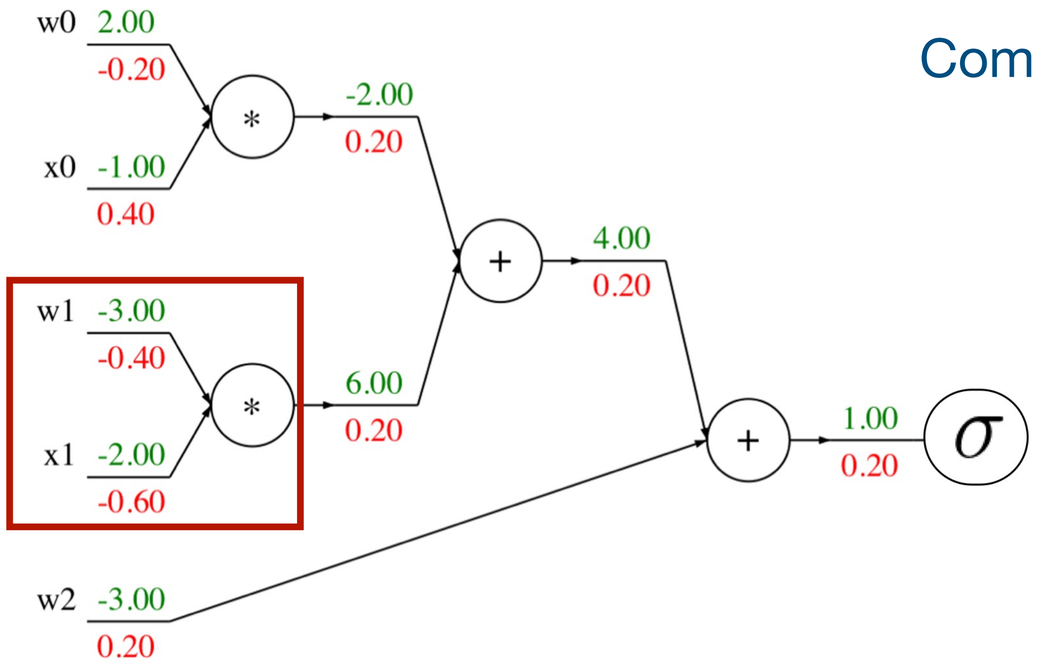
Add

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Backward pass:

Compute gradients

Backprop Implementation: “Flat” gradient code



Forward pass:

Compute outputs

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

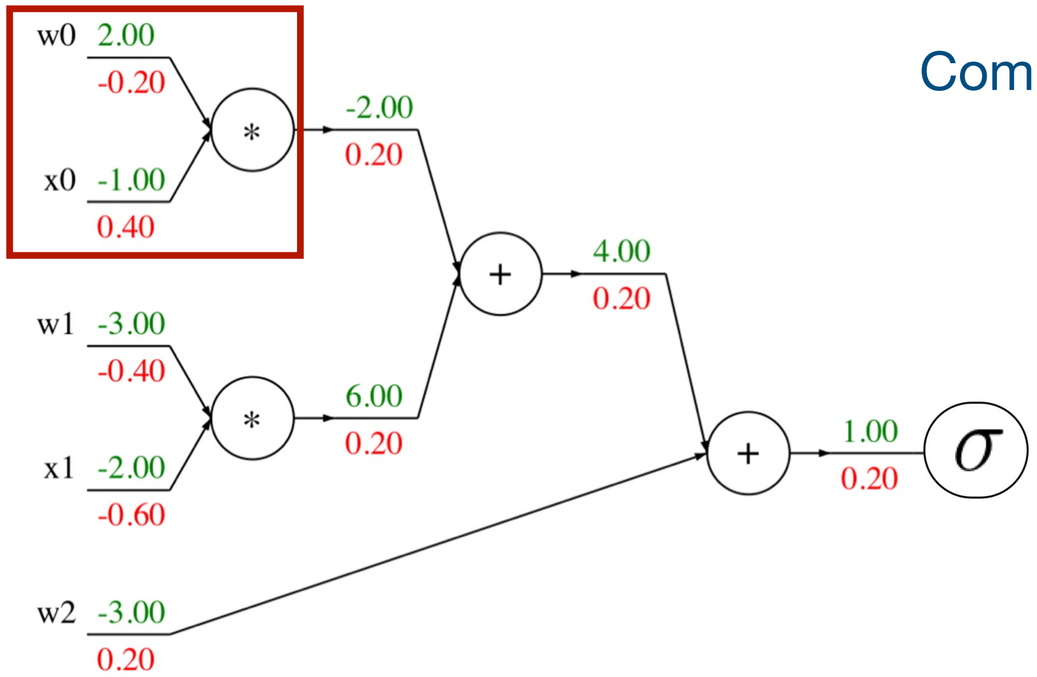
Multiply

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Backward pass:

Compute gradients

Backprop Implementation: “Flat” gradient code



Forward pass:

Compute outputs

```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Multiply

Backward pass:

Compute gradients

“Flat” Backprop: Do this for Project 1 & 2

Forward pass:

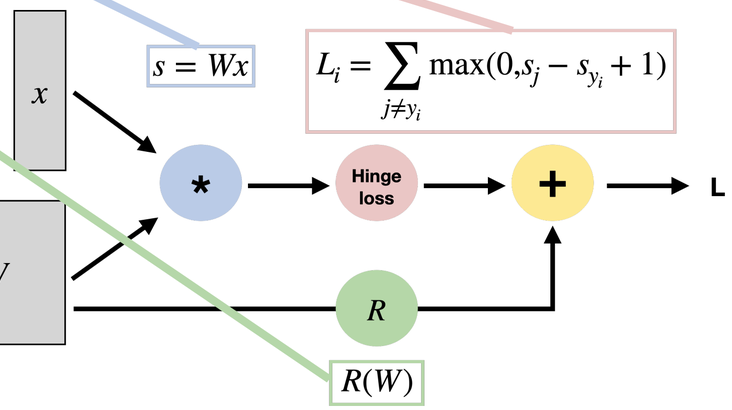
Compute outputs

```
##############################################################################
# TODO:                                                                      #
# Implement a vectorized version of the structured SVM loss, storing the    #
# result in loss.                                                            #
##############################################################################
# Replace "pass" statement with your code
num_classes = W.shape[1]
num_train = X.shape[0]
score = # ...
correct_class_score = # ...
margin = # ...
data_loss = # ...
reg_loss = # ...
loss += data_loss + reg_loss
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```

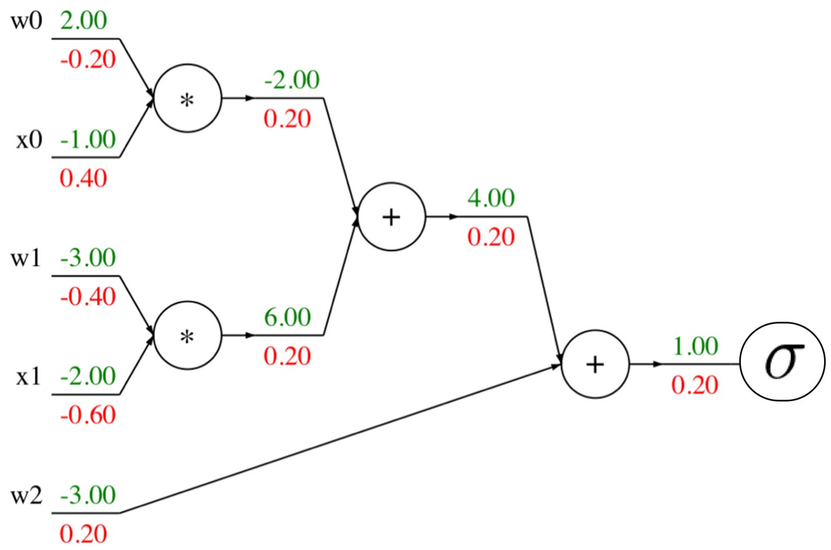
Backward pass:

Compute gradients

```
##############################################################################
# TODO:                                                                      #
# Implement a vectorized version of the gradient for the structured SVM     #
# loss, storing the result in dW.                                            #
#                                                                            #
# Hint: Instead of computing the gradient from scratch, it may be easier     #
# to reuse some of the intermediate values that you used to compute the      #
# loss.                                                                      #
##############################################################################
# Replace "pass" statement with your code
dmargins = # ...
dscores = # ...
dW = # ...
##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
```



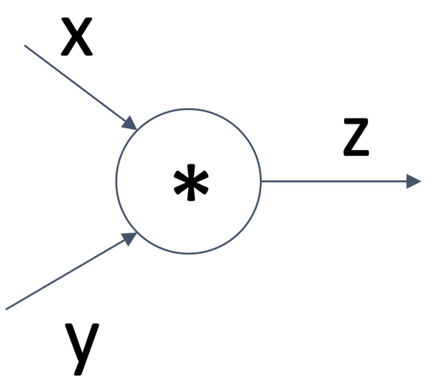
Backprop Implementation: Modular API



Graph (or Net) object *(rough pseudo code)*

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Example: PyTorch Autograd Functions



(x,y,z are scalars)

```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to stash some values for use in backward

Upstream gradient

Multiply upstream and local gradients

Scalar Derivatives

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Vector Derivatives

$x \in \mathbb{R}, y \in \mathbb{R}$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

$x \in \mathbb{R}^N, y \in \mathbb{R}$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N,$$

$$\left(\frac{\partial y}{\partial x}\right)_i = \frac{\partial y}{\partial x_i}$$

For each element of x, if it changes by a small amount then how much will y change?

Vector Derivatives

$x \in \mathbb{R}, y \in \mathbb{R}$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

$x \in \mathbb{R}^N, y \in \mathbb{R}$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N,$$

$$\left(\frac{\partial y}{\partial x}\right)_i = \frac{\partial y}{\partial x_i}$$

For each element of x, if it changes by a small amount then how much will y change?

$x \in \mathbb{R}^N, y \in \mathbb{R}^M$

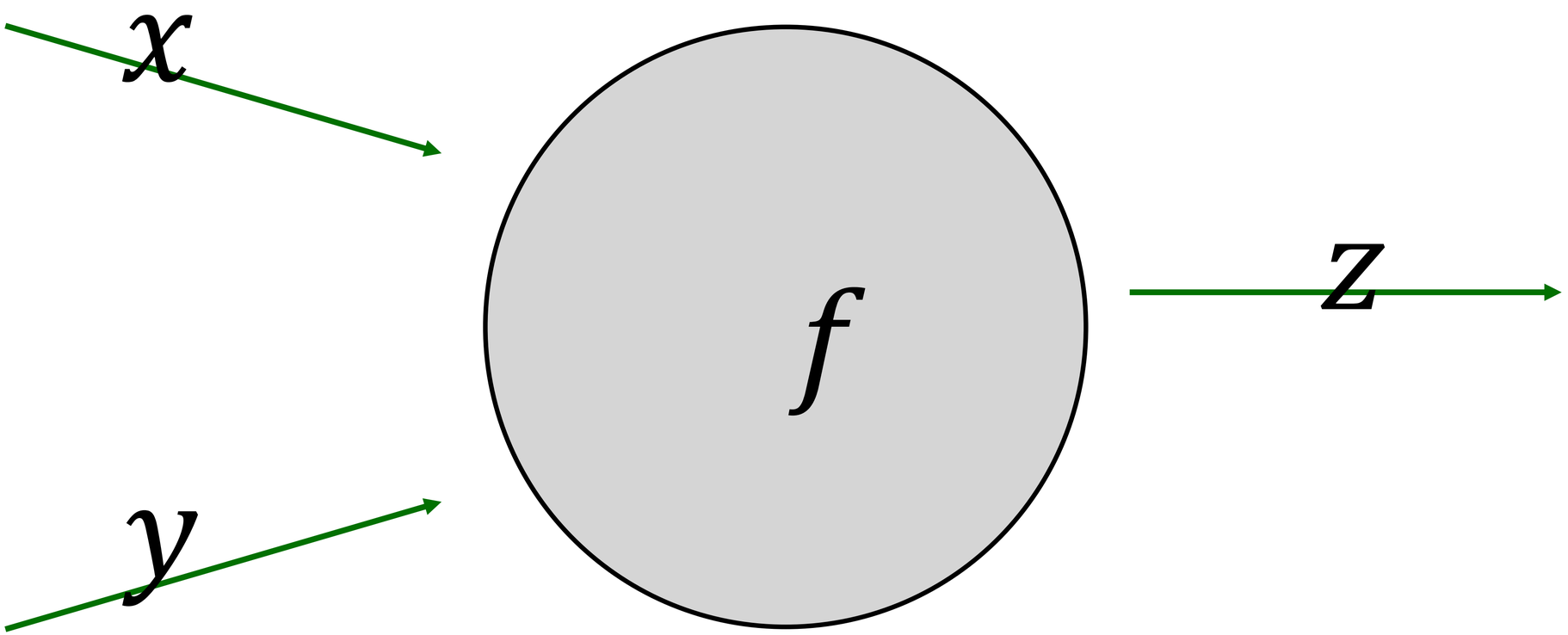
Derivative is **Jacobian**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M}$$

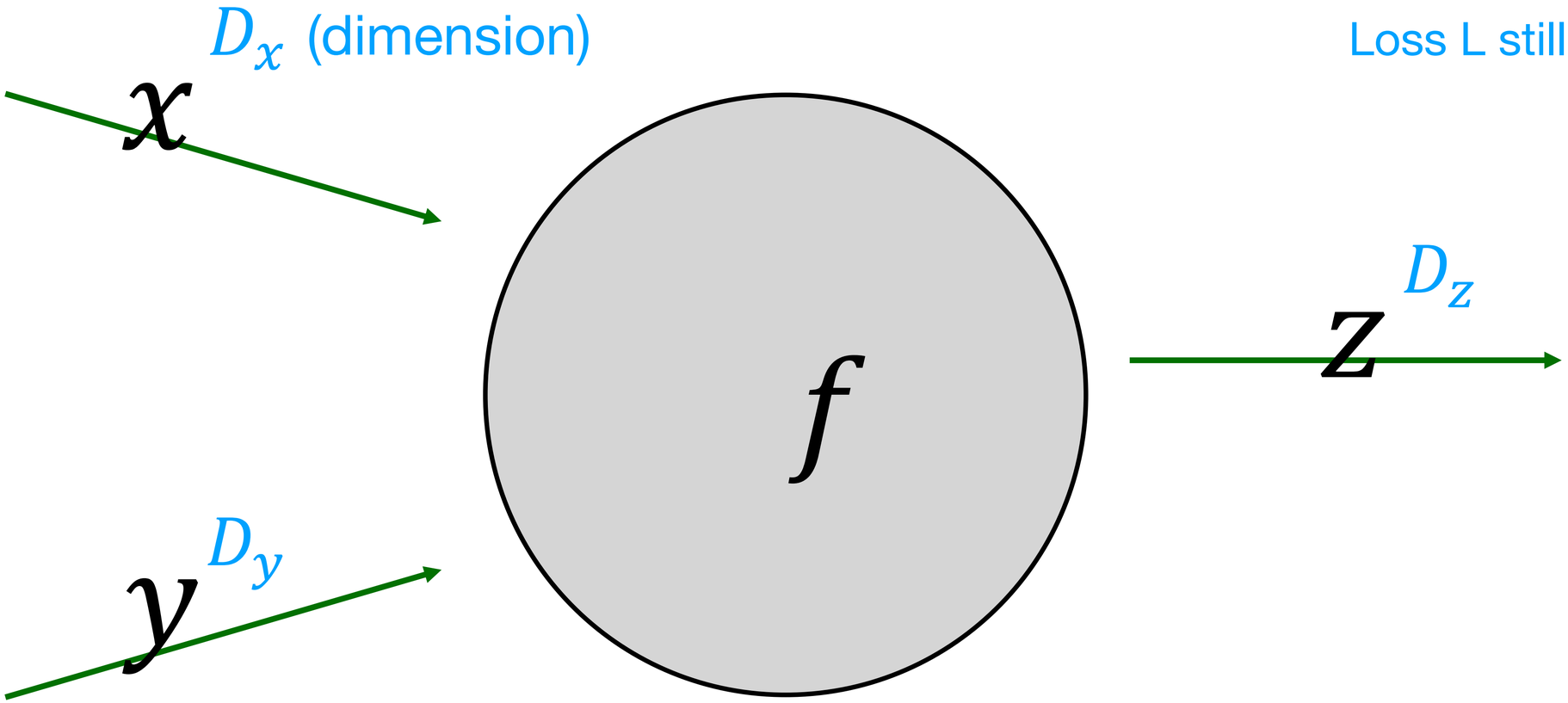
$$\left(\frac{\partial y}{\partial x}\right)_{i,j} = \frac{\partial y_j}{\partial x_i}$$

For each element of x, if it changes by a small amount then how much will each element of y change?

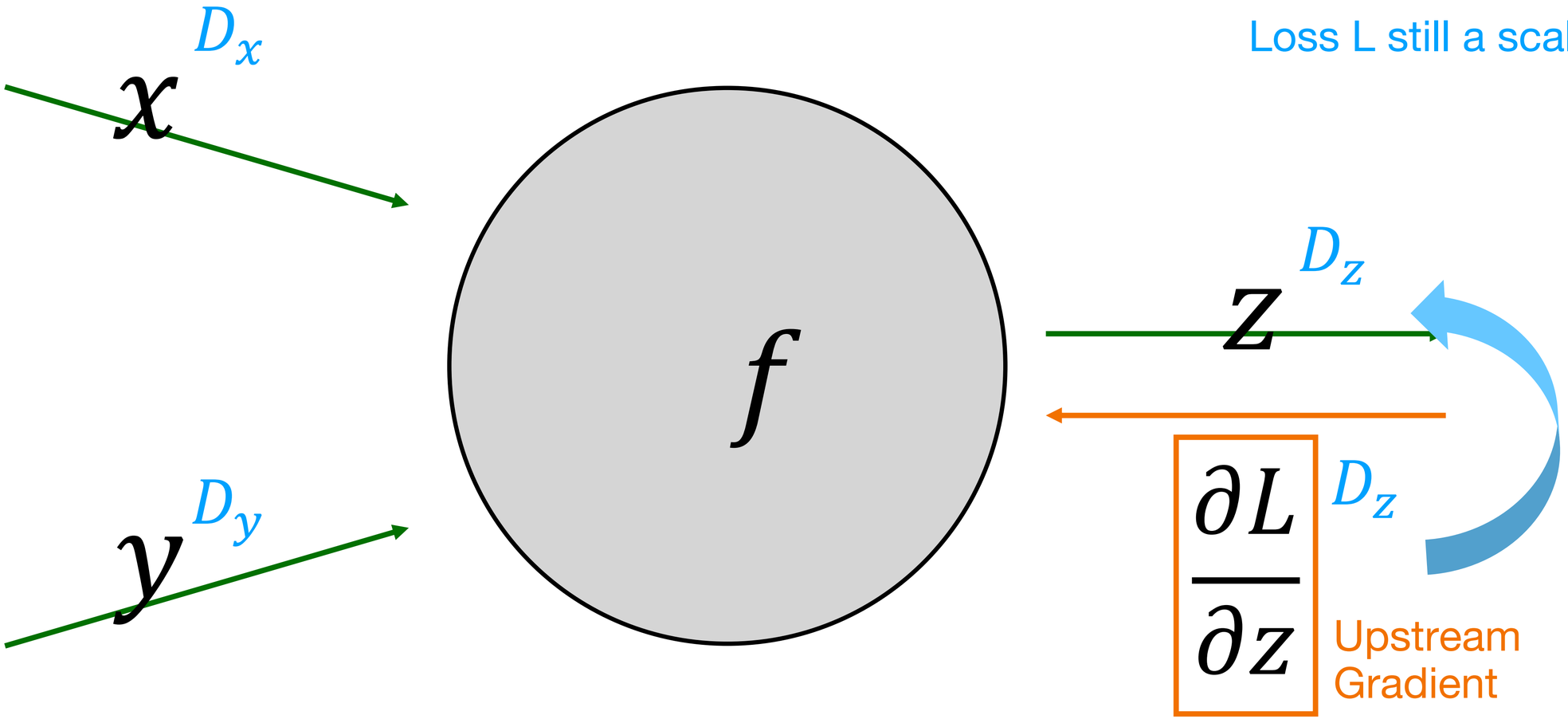
Backprop with Vectors: Dimensions



Backprop with Vectors: Dimensions



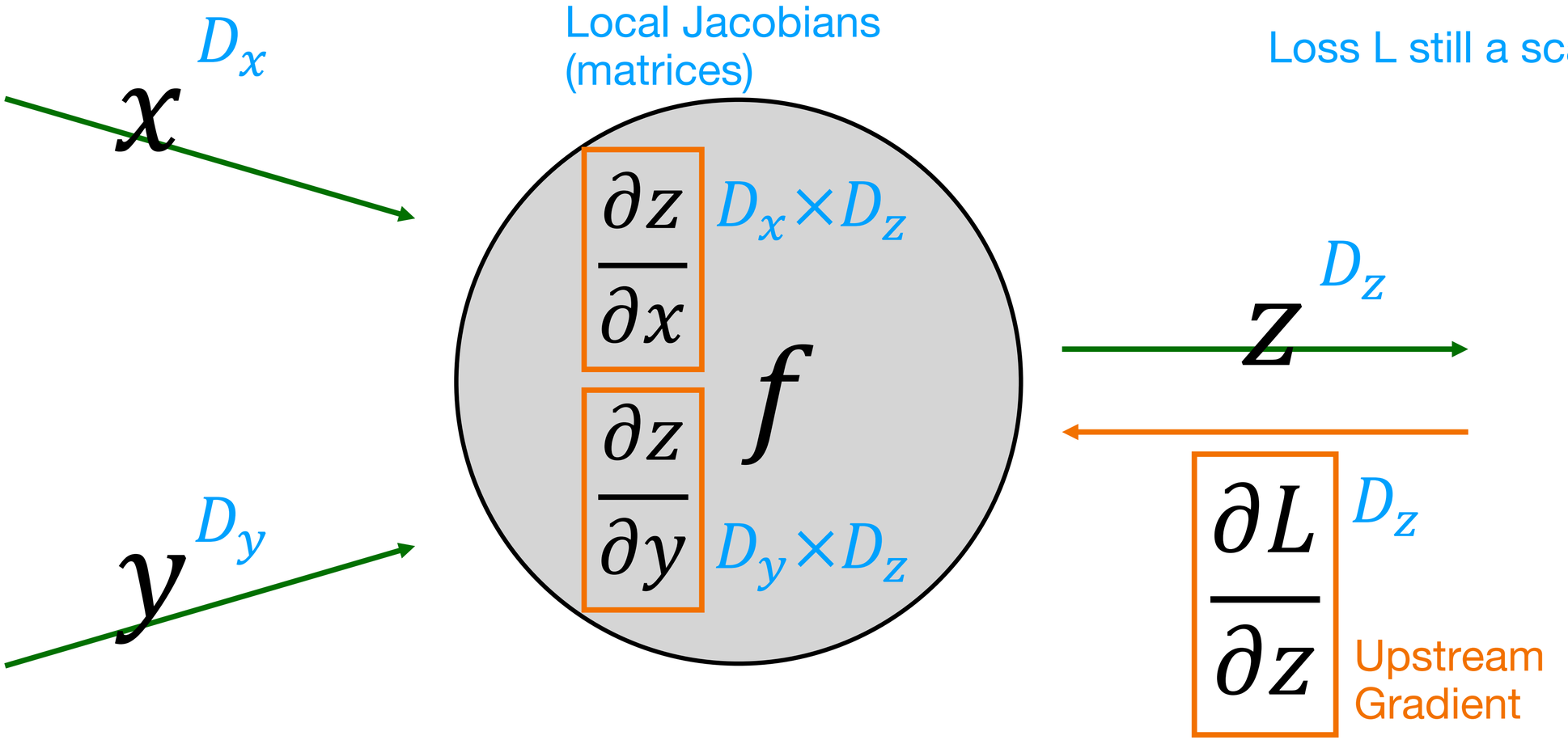
Backprop with Vectors: Dimensions



Loss L still a scalar!

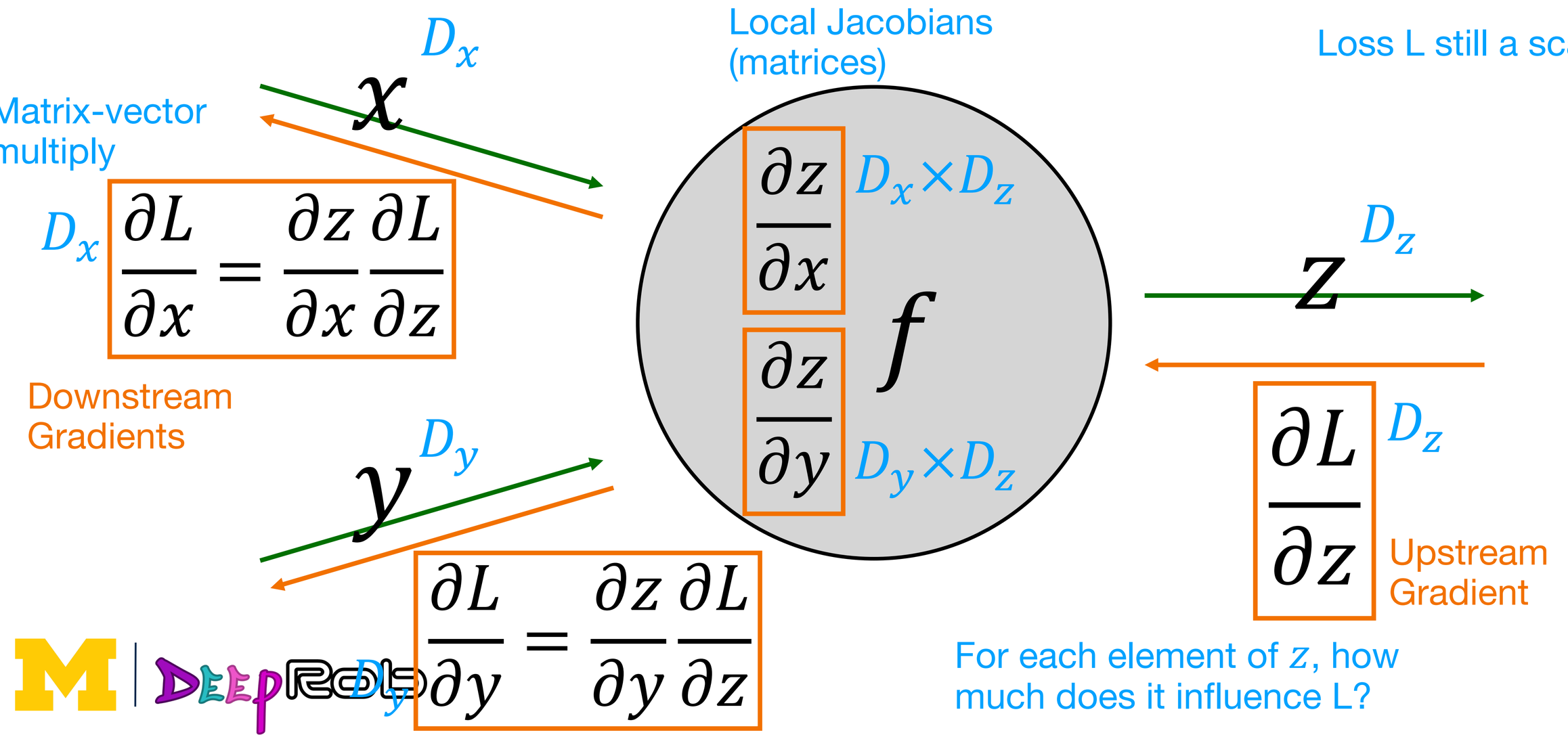
For each element of z, how much does it influence L?

Backprop with Vectors: Dimensions

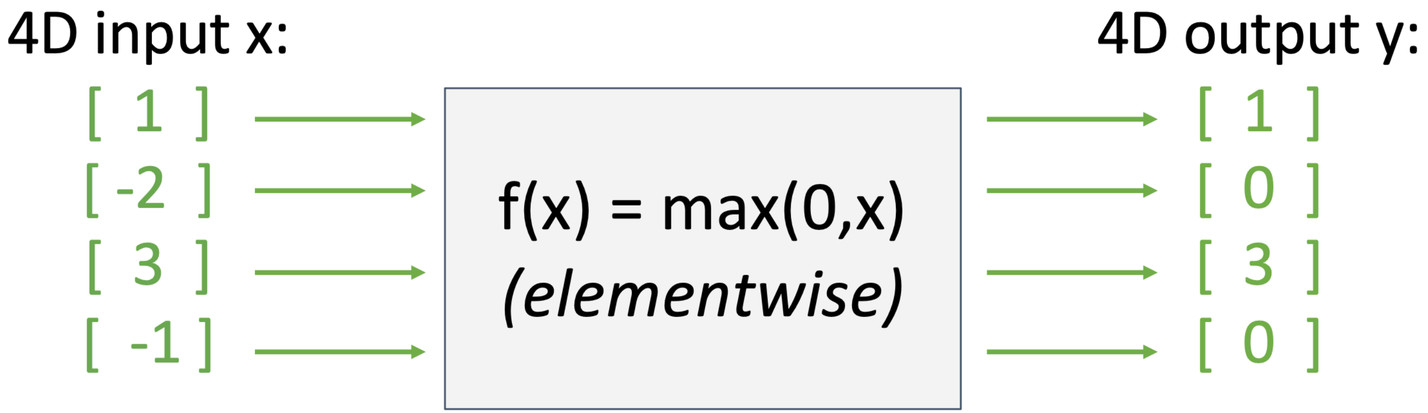


For each element of z, how much does it influence L?

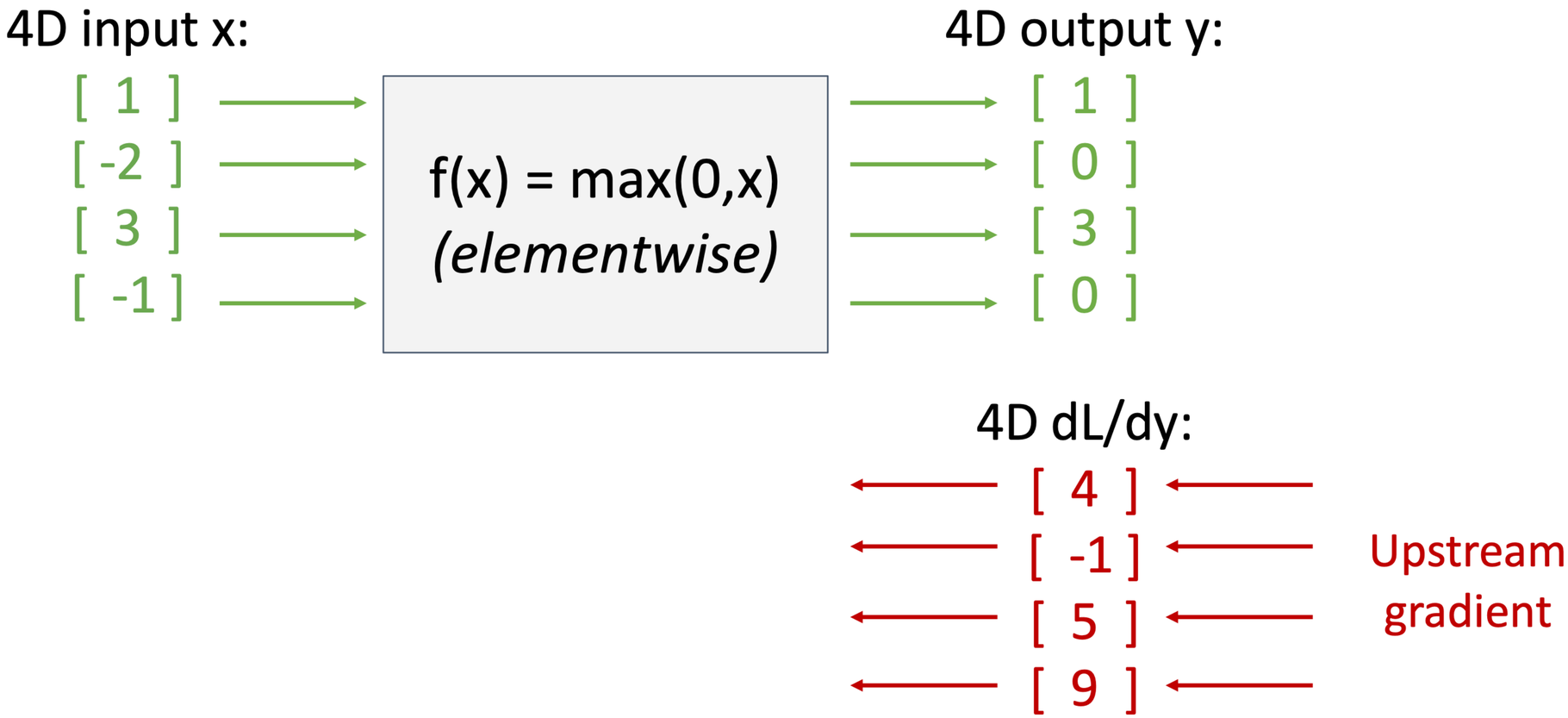
Backprop with Vectors: Dimensions



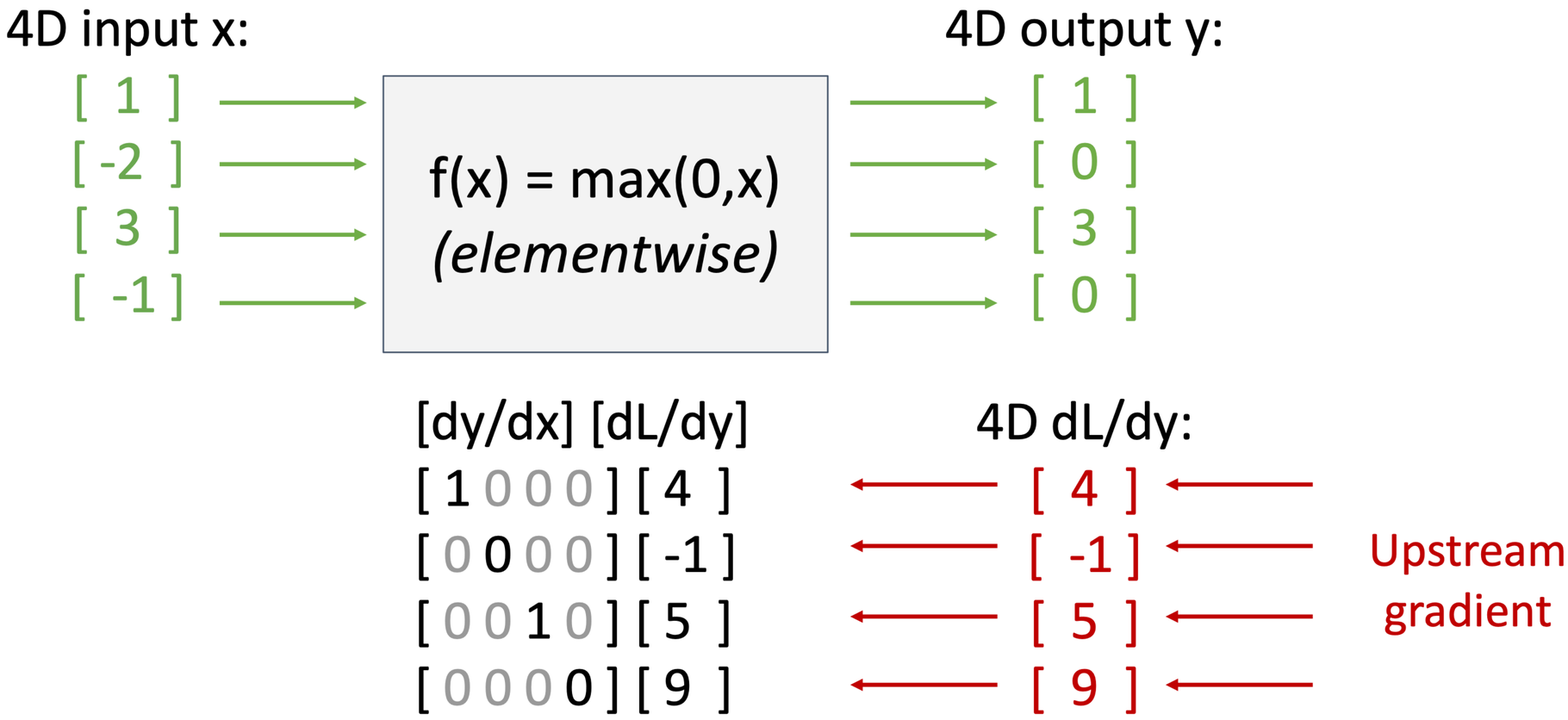
Backprop with Vectors



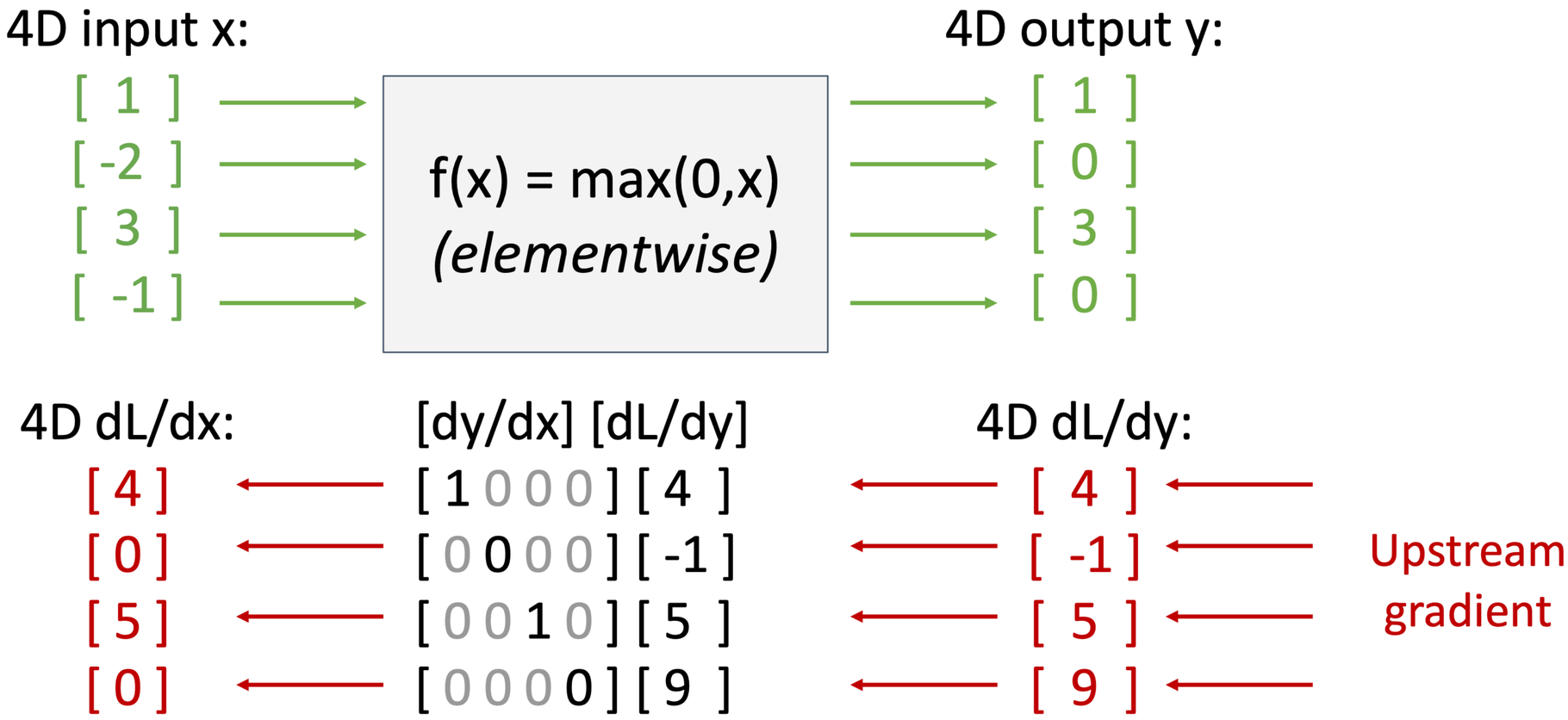
Backprop with Vectors



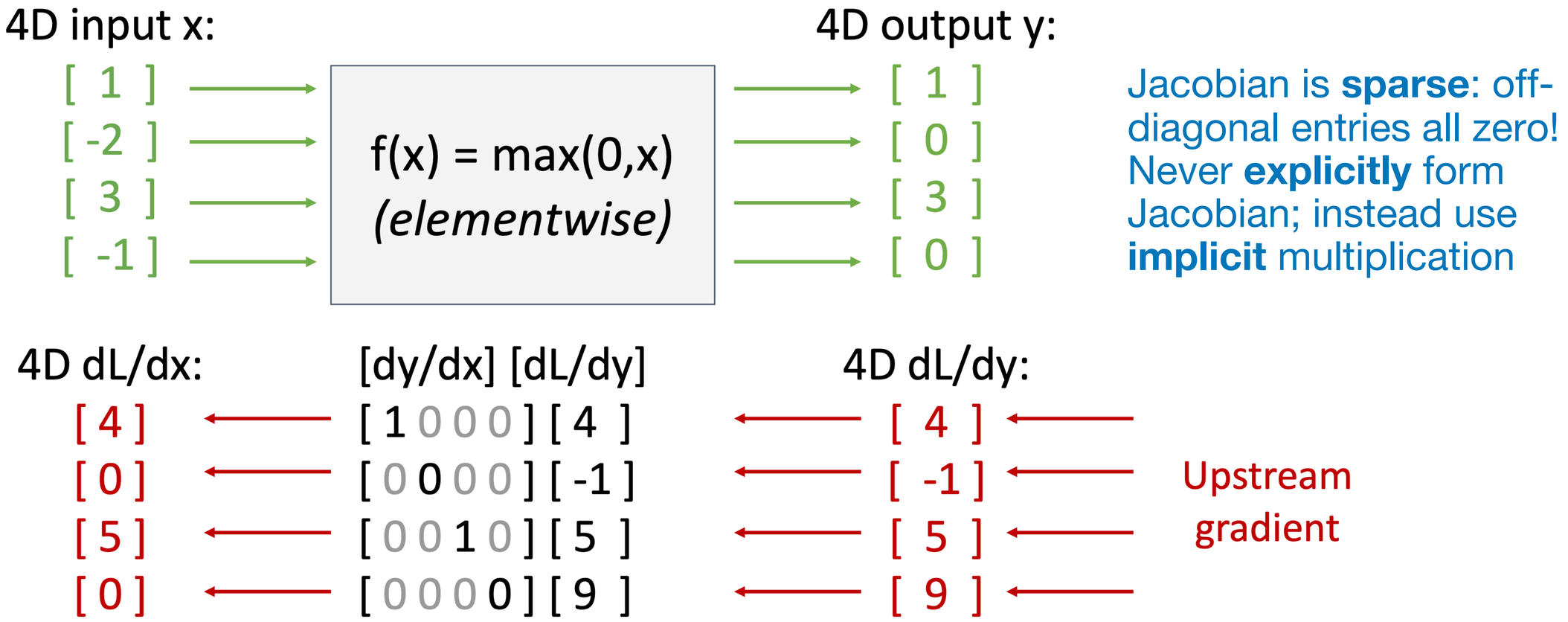
Backprop with Vectors



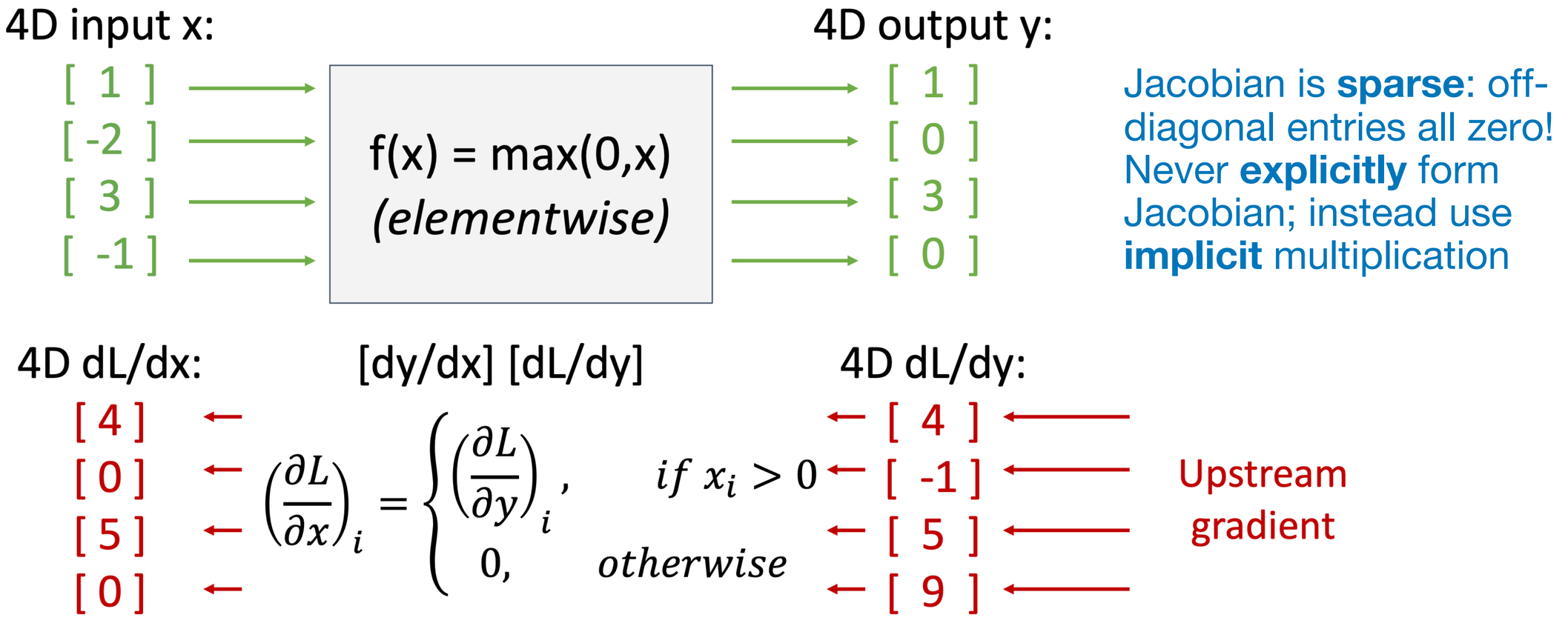
Backprop with Vectors



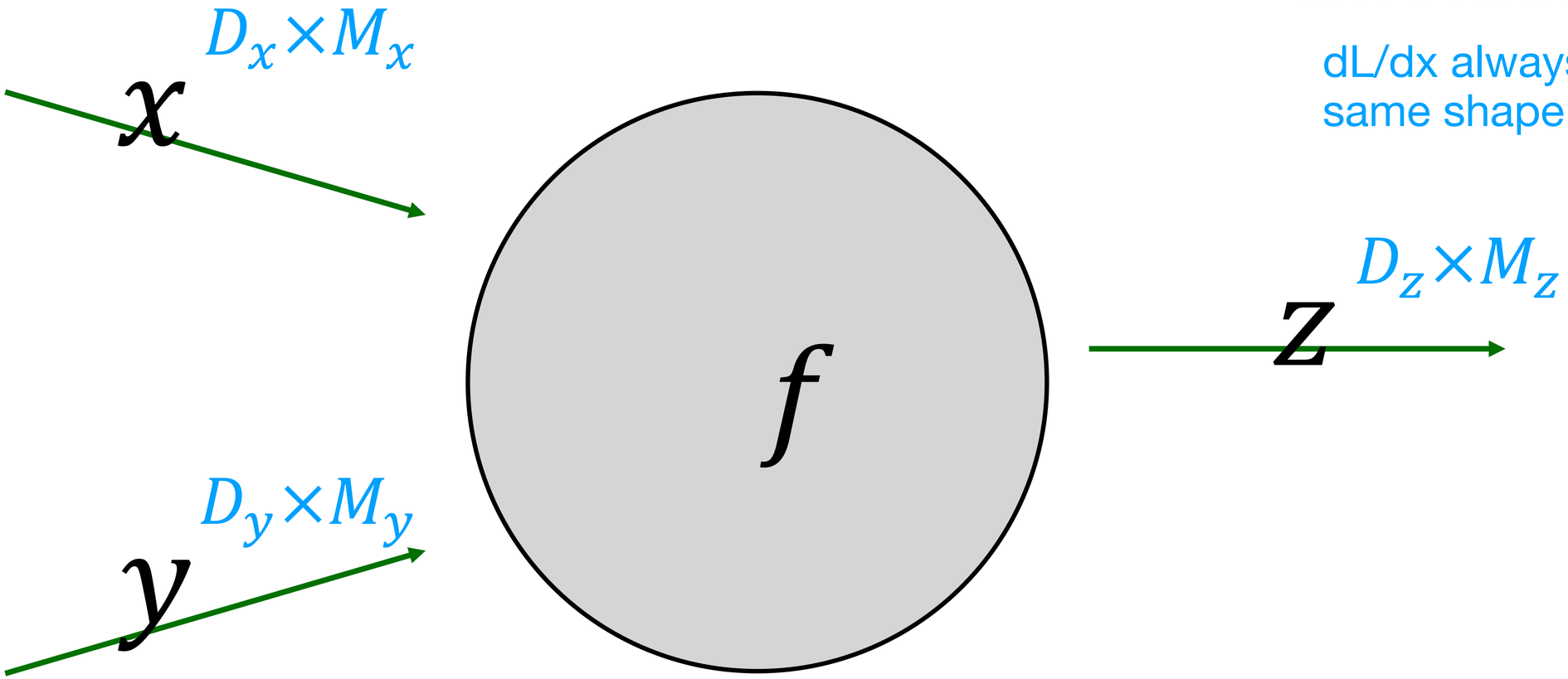
Backprop with Vectors



Backprop with Vectors



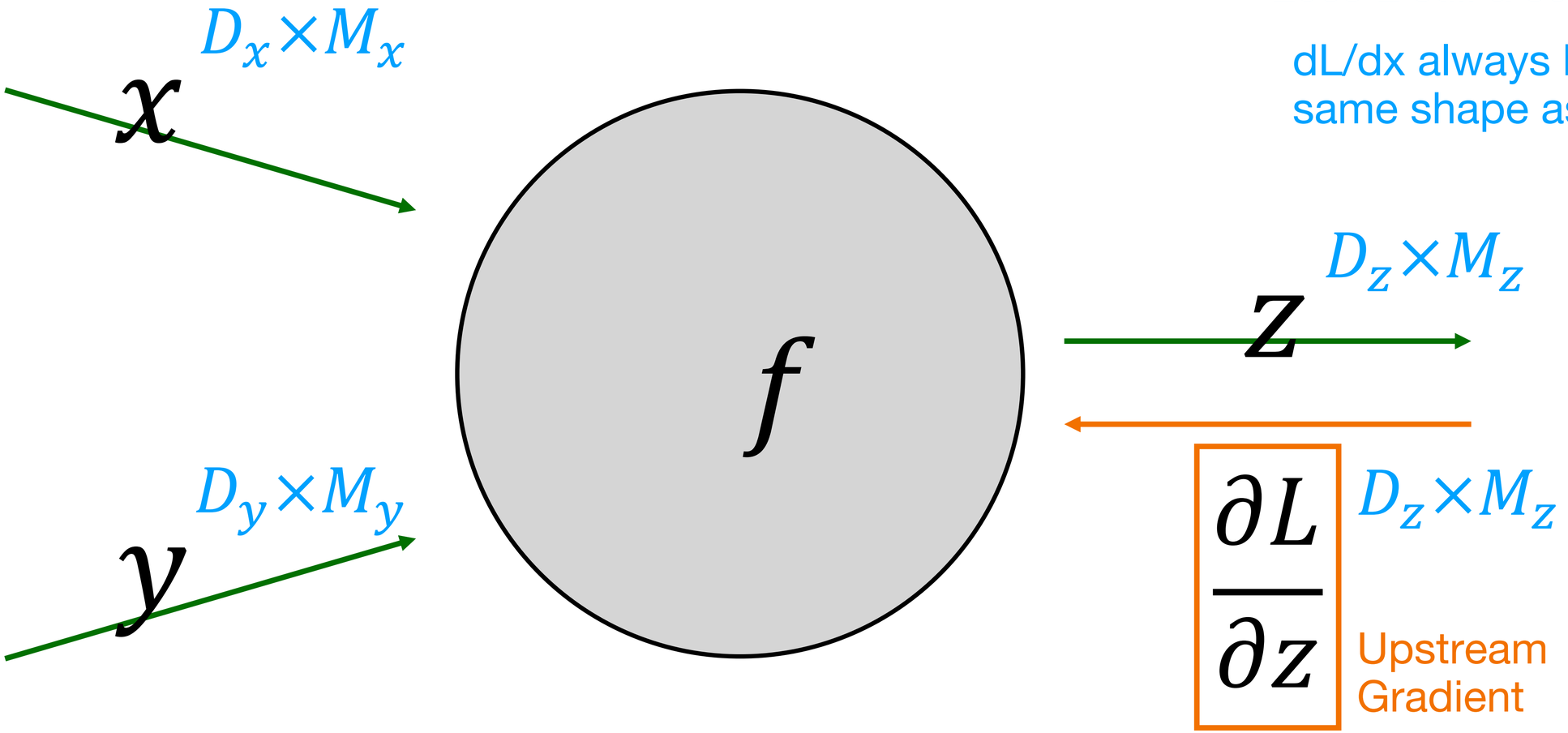
Backprop with Matrices (or Tensors)



Loss L still a scalar!

dL/dx always has the same shape as x!

Backprop with Matrices (or Tensors)

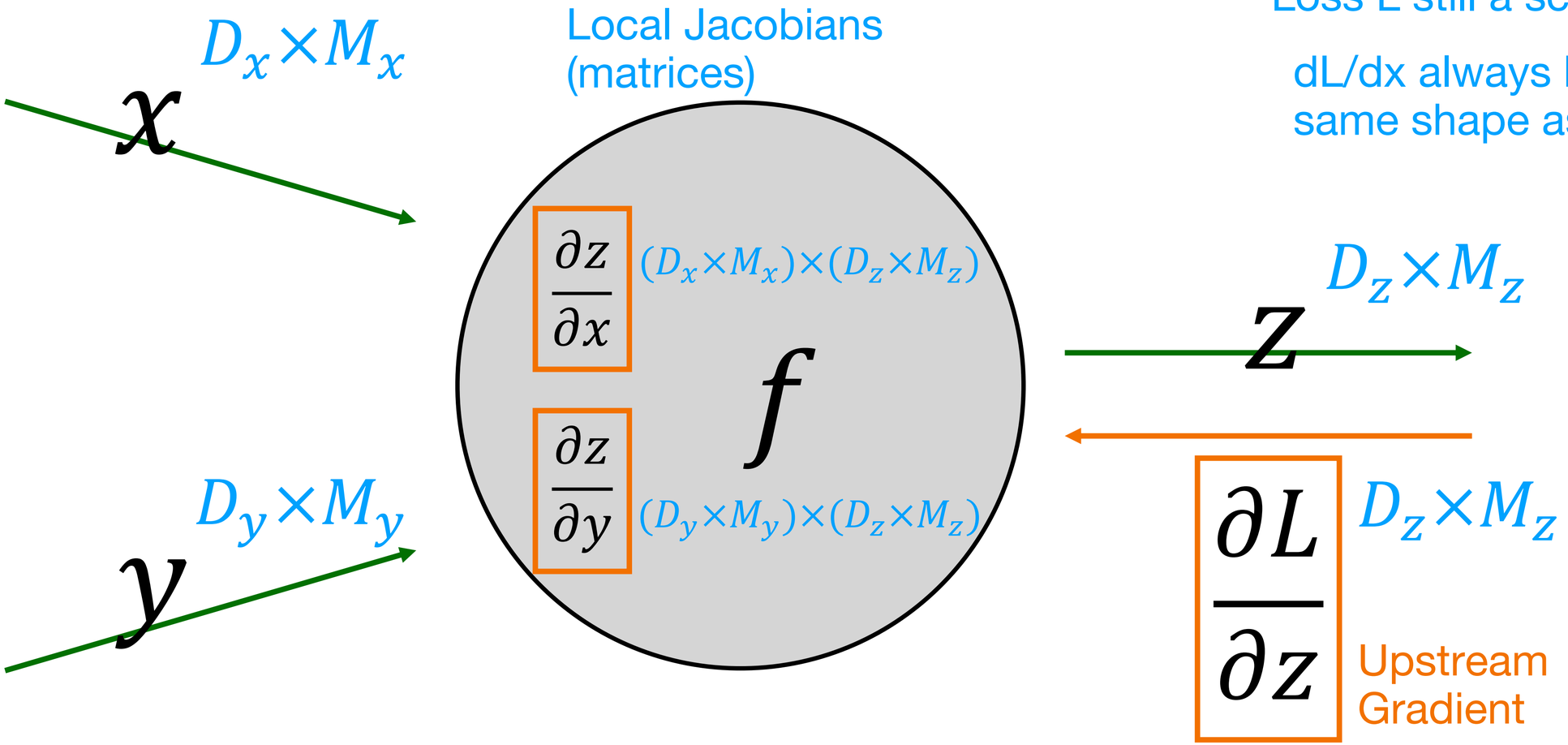


Loss L still a scalar!

dL/dx always has the same shape as x!

For each element of z, how much does it influence L?

Backprop with Matrices (or Tensors)

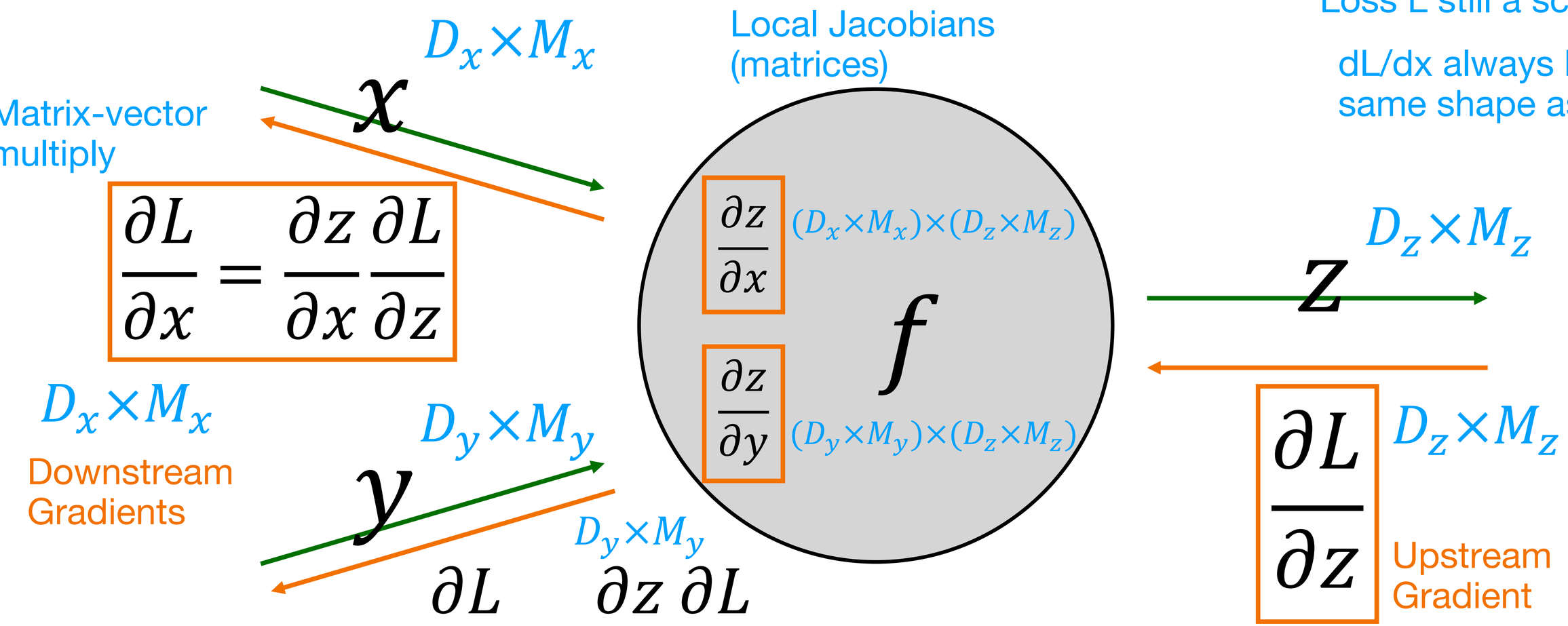


Loss L still a scalar!

dL/dx always has the same shape as x!

For each element of z, how much does it influence L?

Backprop with Matrices (or Tensors)



For each element of z, how much does it influence L?

Example: Matrix Multiplication

x: [N×D]

[2 1 -3]
[-3 4 2]

w: [D×M]

[3 2 1 -1]
[2 1 3 2]
[3 2 1 -2]

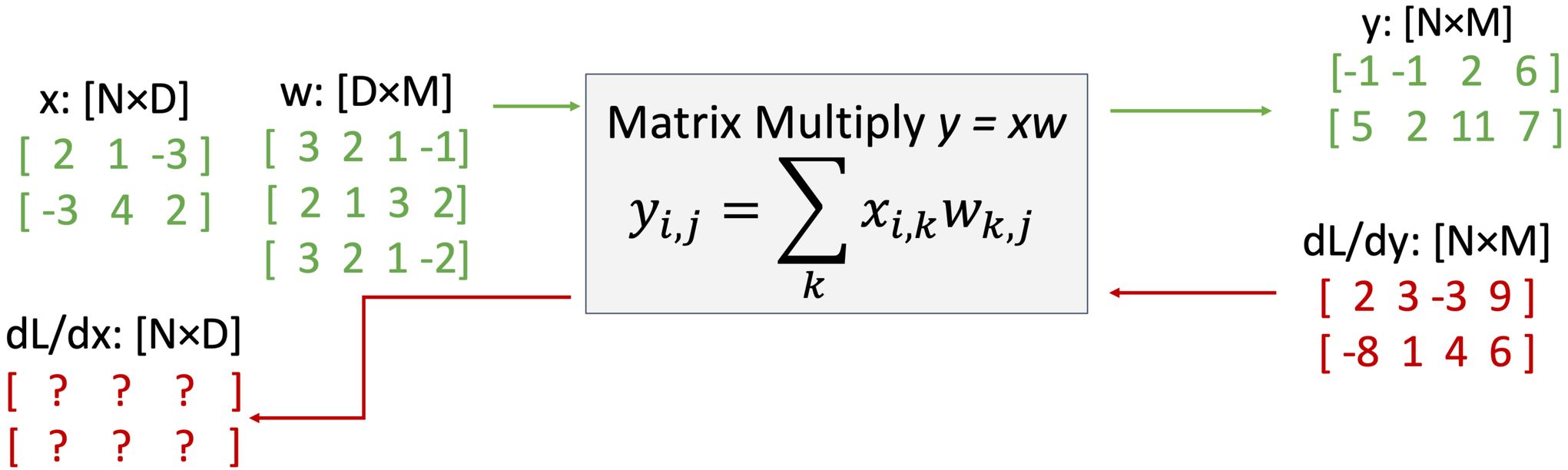
Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

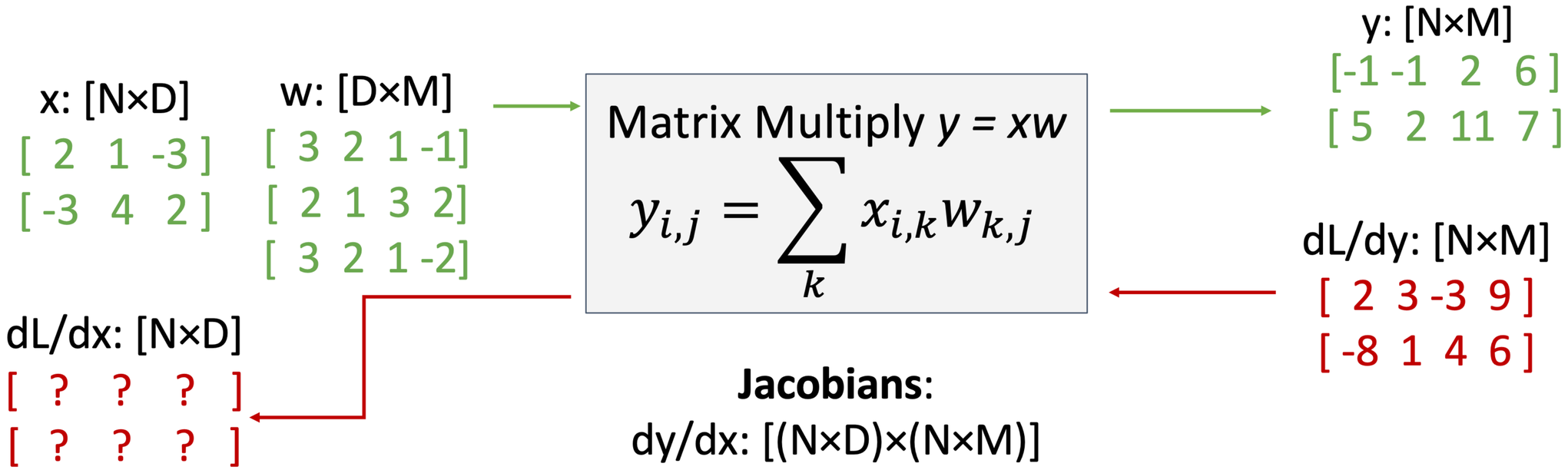
y: [N×M]

[-1 -1 2 6]
[5 2 11 7]

Example: Matrix Multiplication



Example: Matrix Multiplication



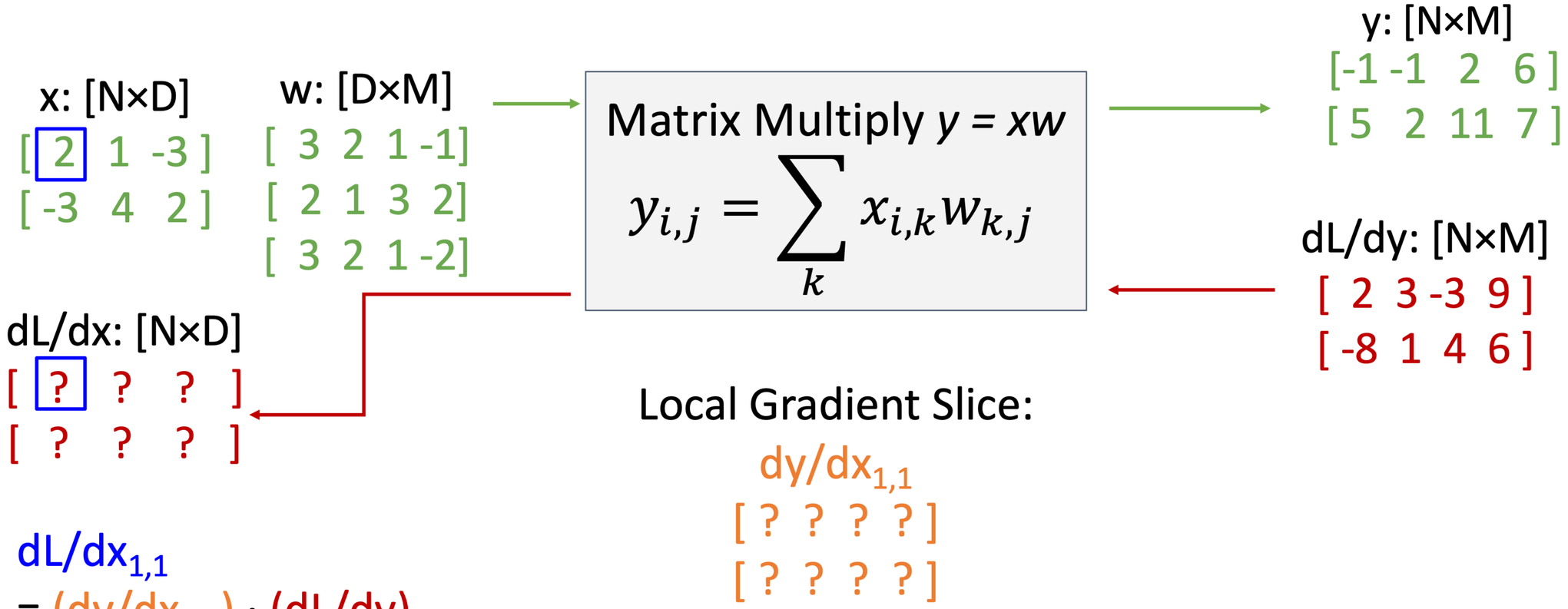
Jacobians:

dy/dx: [(N×D)×(N×M)]

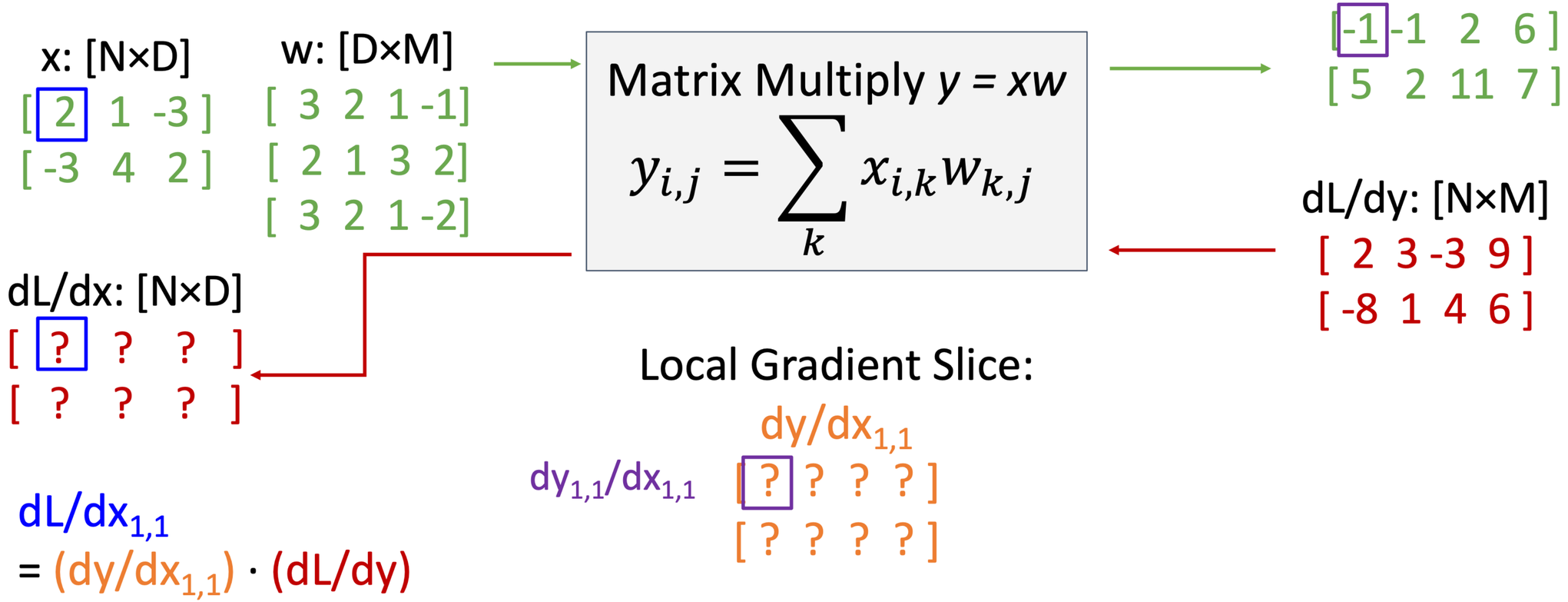
dy/dw: [(D×M)×(N×M)]

For a neural net we may have
N=64, D=M=4096
Each Jacobian takes 256 GB of memory! Must work with them implicitly!

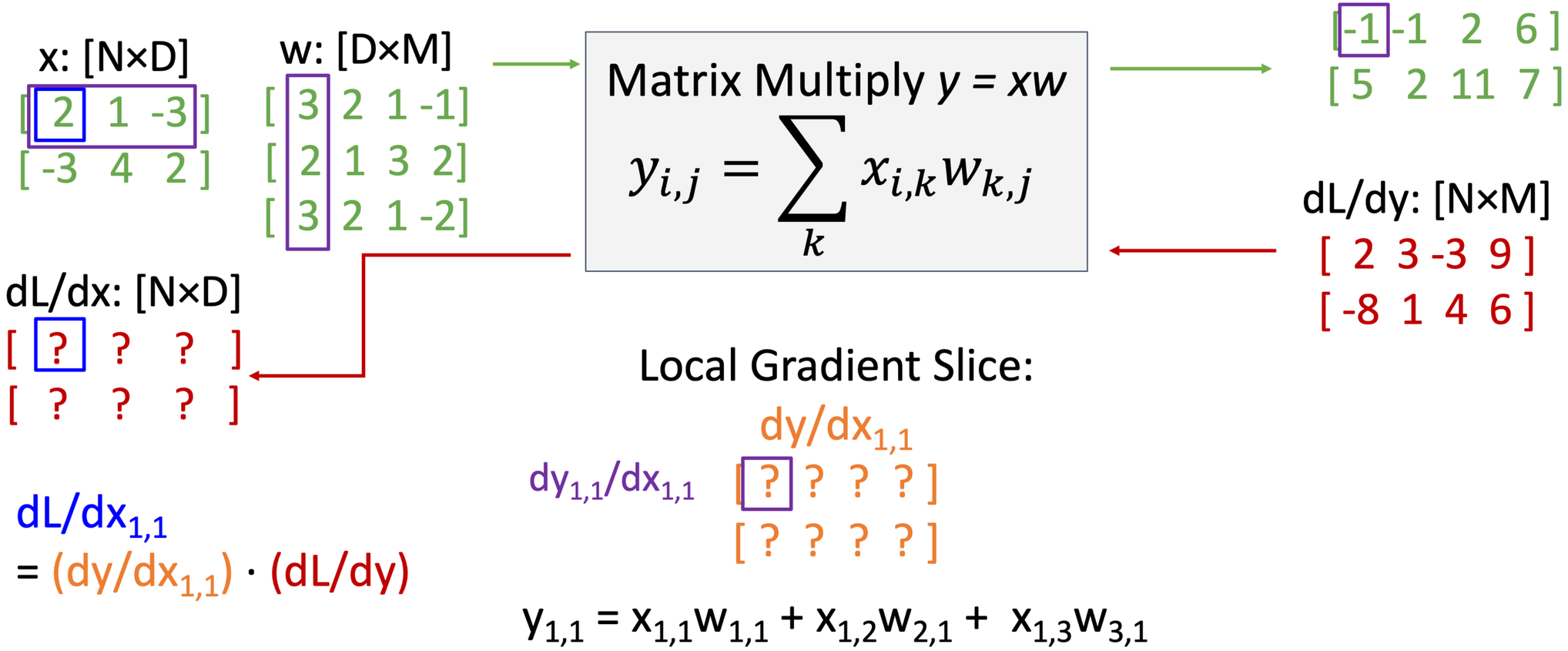
Example: Matrix Multiplication



Example: Matrix Multiplication



Example: Matrix Multiplication



Example: Matrix Multiplication

x: [N×D]

[2 1 -3]
[-3 4 2]

w: [D×M]

[3 2 1 -1]
[2 1 3 2]
[3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6]
[5 2 11 7]

dL/dy: [N×M]

[2 3 -3 9]
[-8 1 4 6]

dL/dx: [N×D]

[? ? ?]
[? ? ?]

$dL/dx_{1,1}$
$= (dy/dx_{1,1}) \cdot (dL/dy)$

Local Gradient Slice:

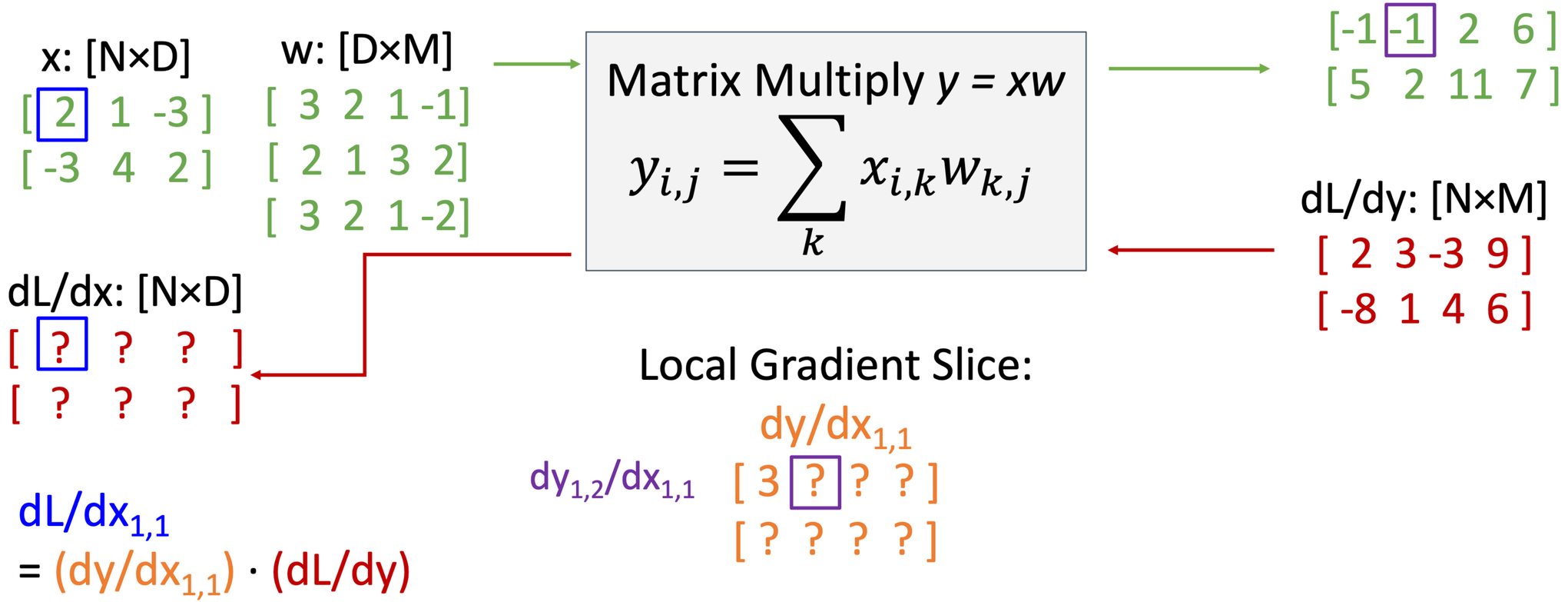
$dy/dx_{1,1}$

$dy_{1,1}/dx_{1,1}$ [3 ? ? ?]
[? ? ? ?]

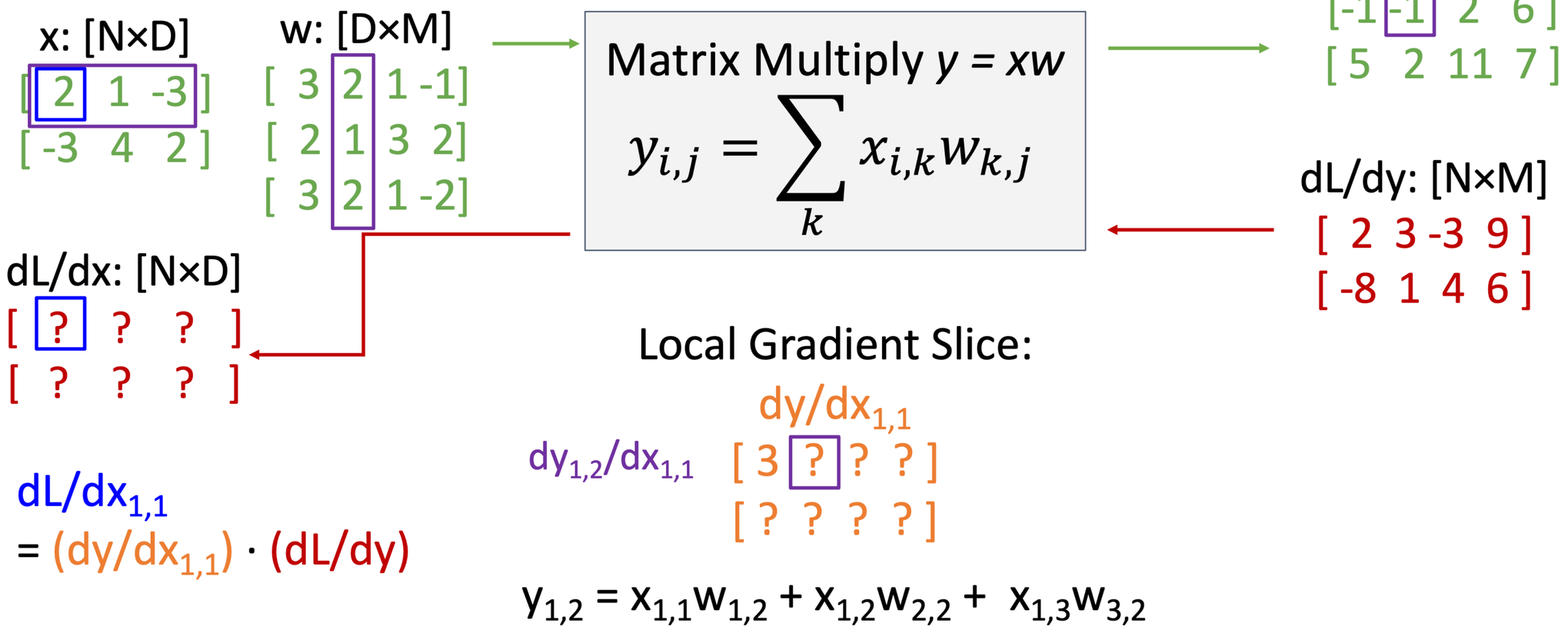
$y_{1,1} = x_{1,1}w_{1,1} + x_{1,2}w_{2,1} + x_{1,3}w_{3,1}$

$\Rightarrow dy_{1,1}/dx_{1,1} = w_{1,1}$

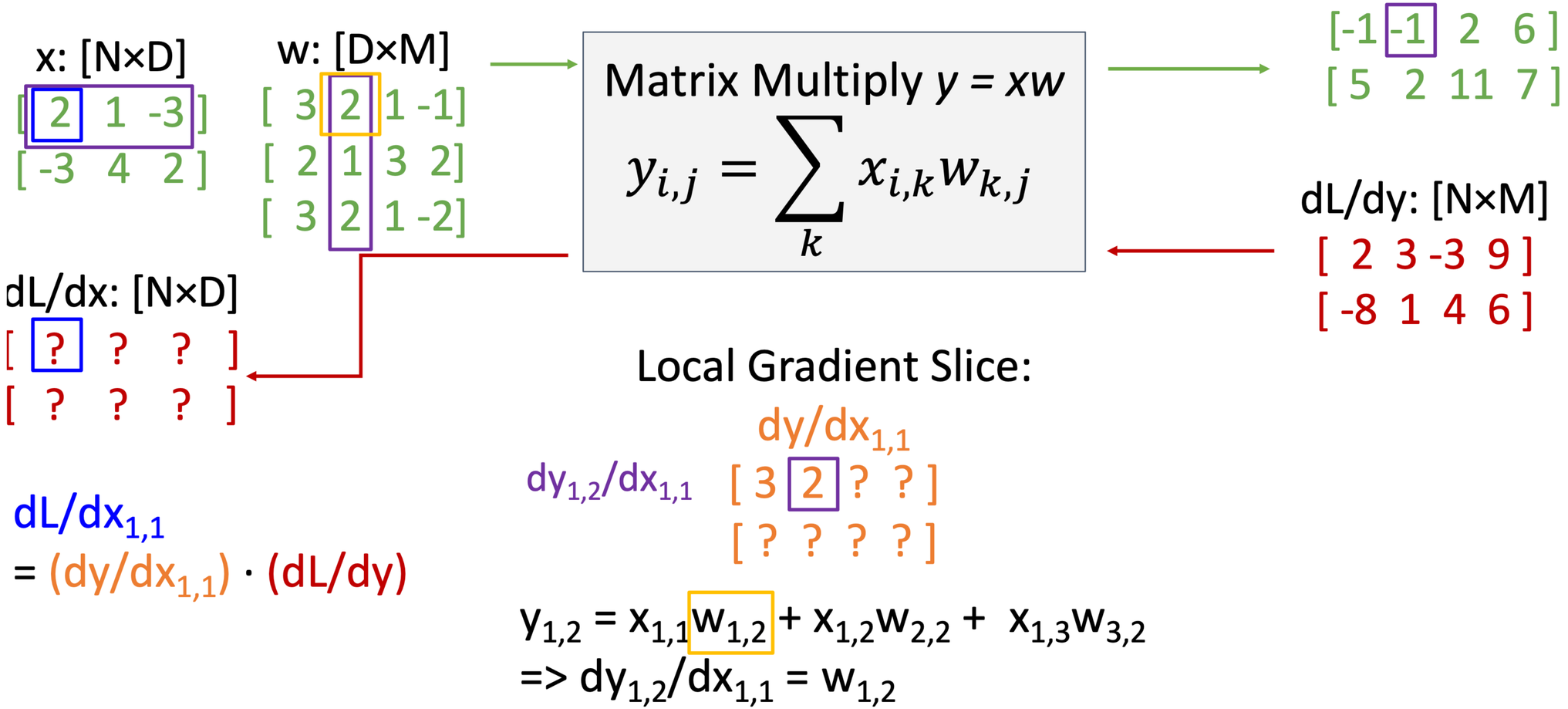
Example: Matrix Multiplication



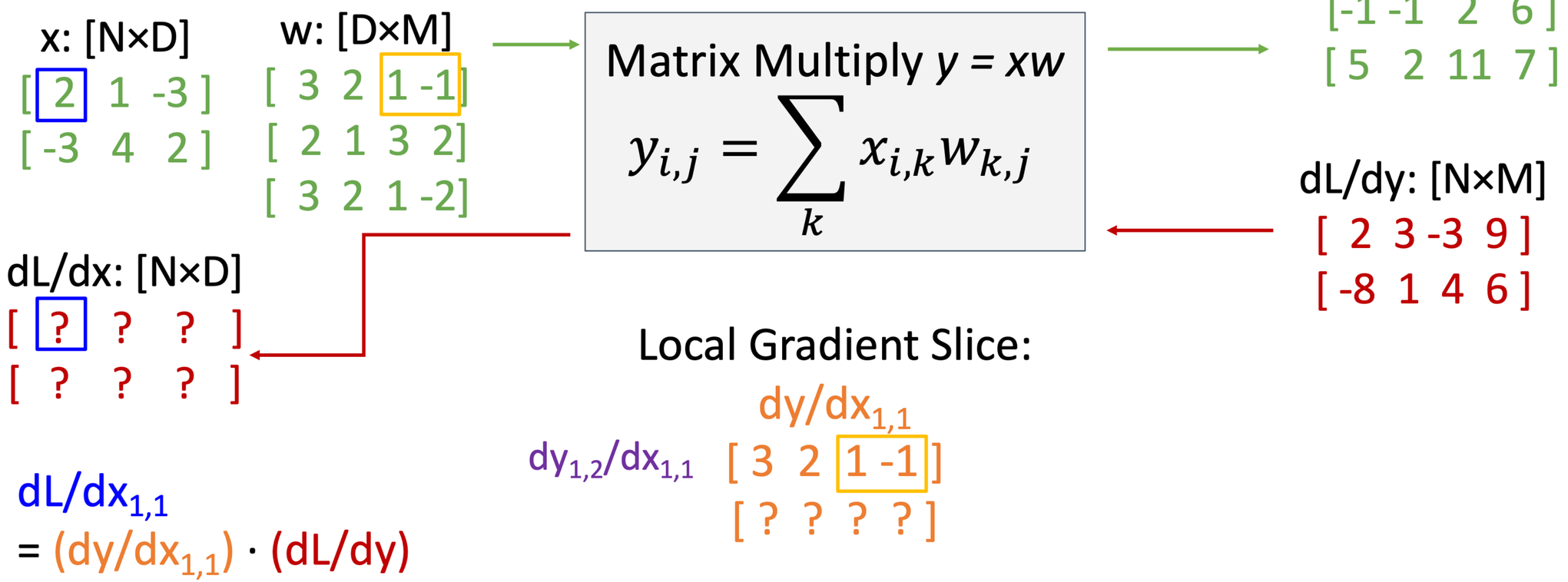
Example: Matrix Multiplication



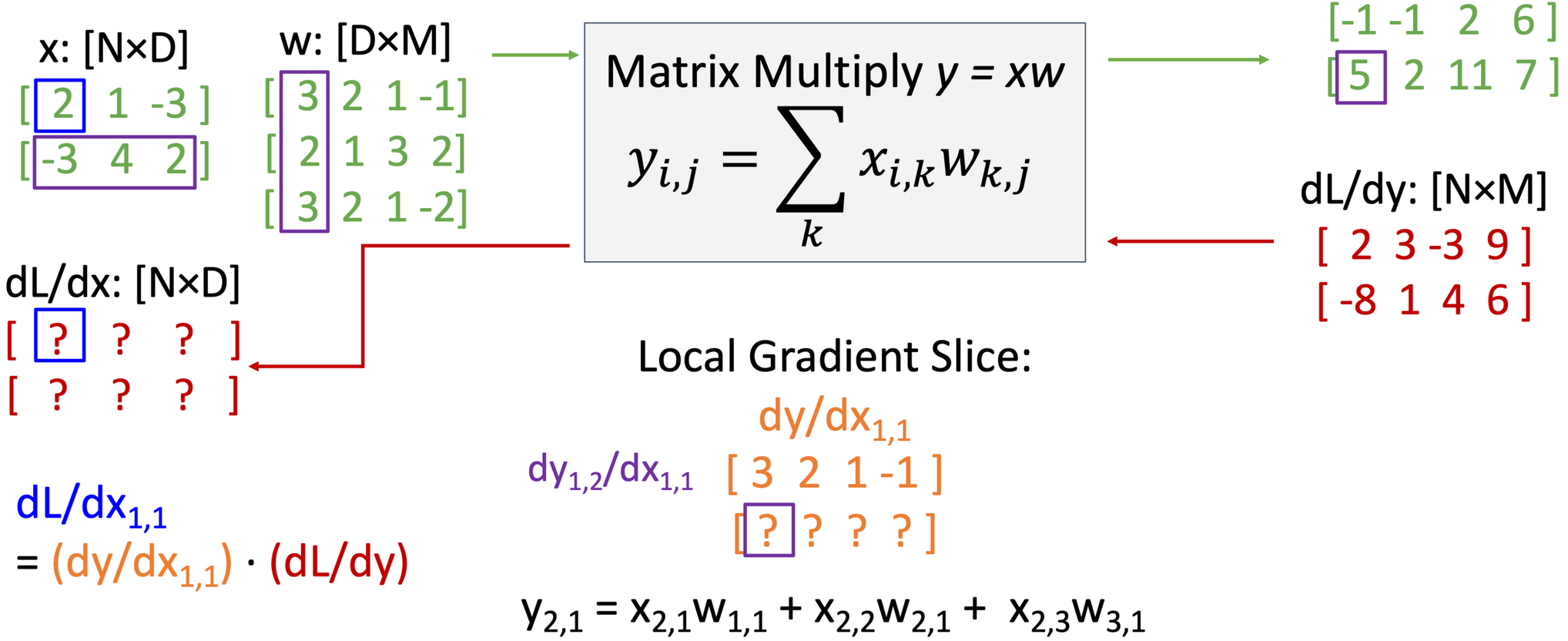
Example: Matrix Multiplication



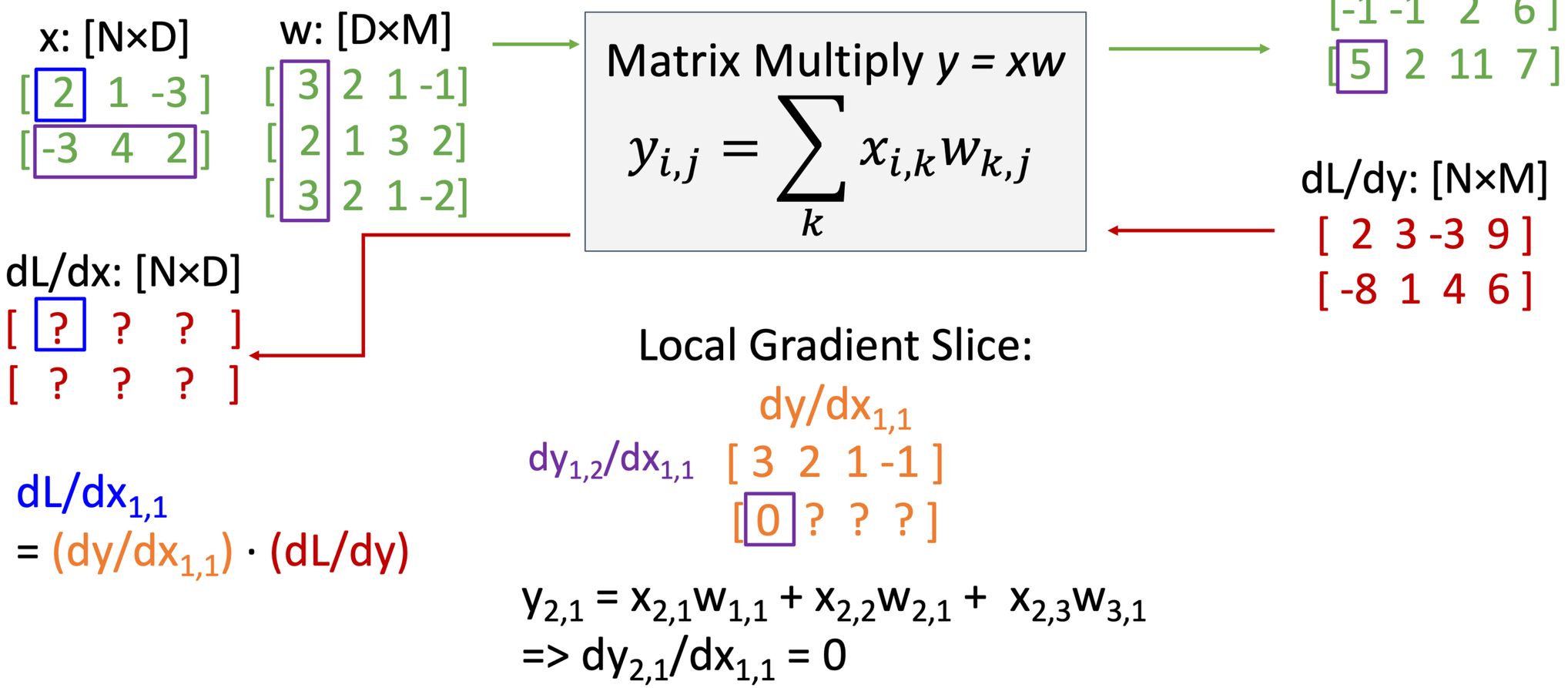
Example: Matrix Multiplication



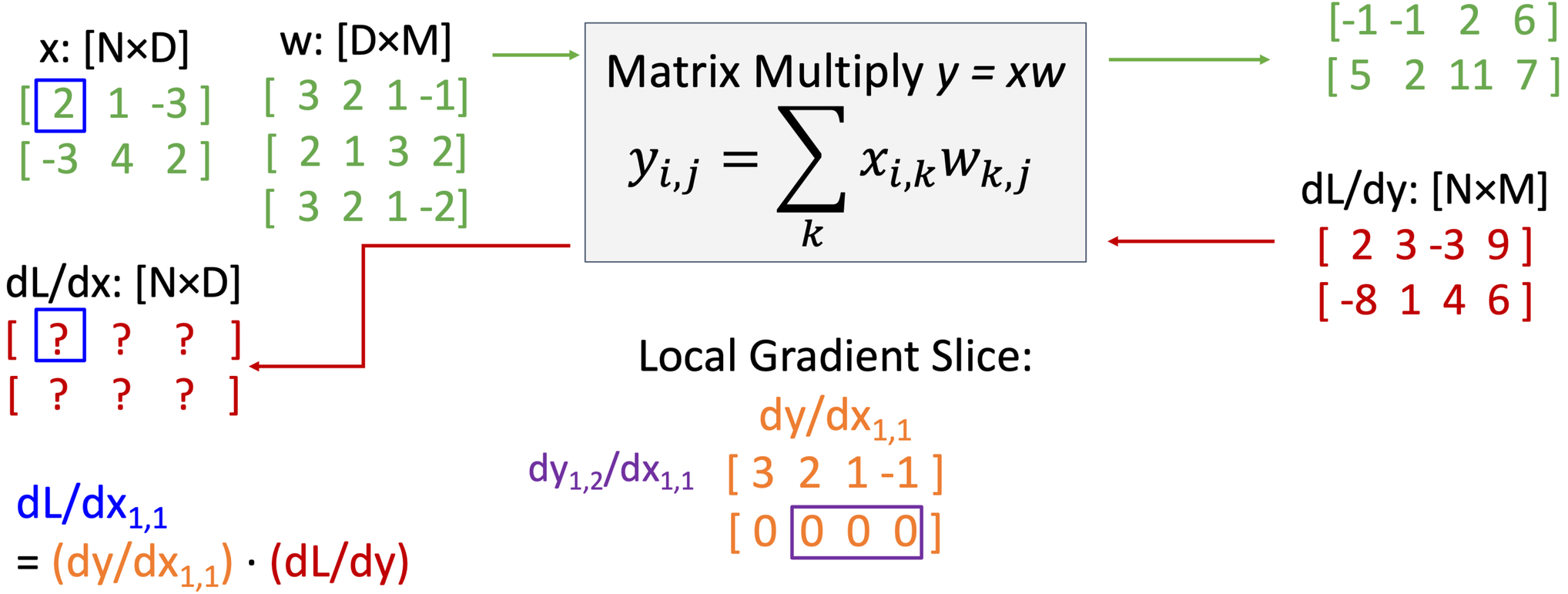
Example: Matrix Multiplication



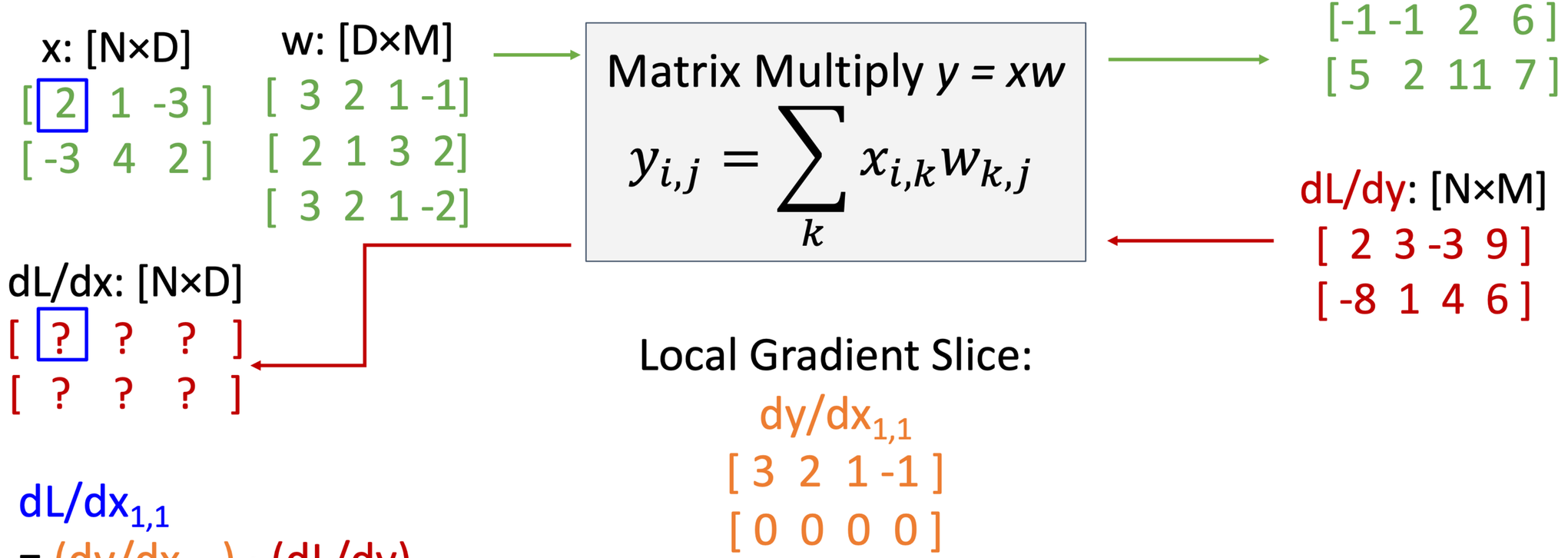
Example: Matrix Multiplication



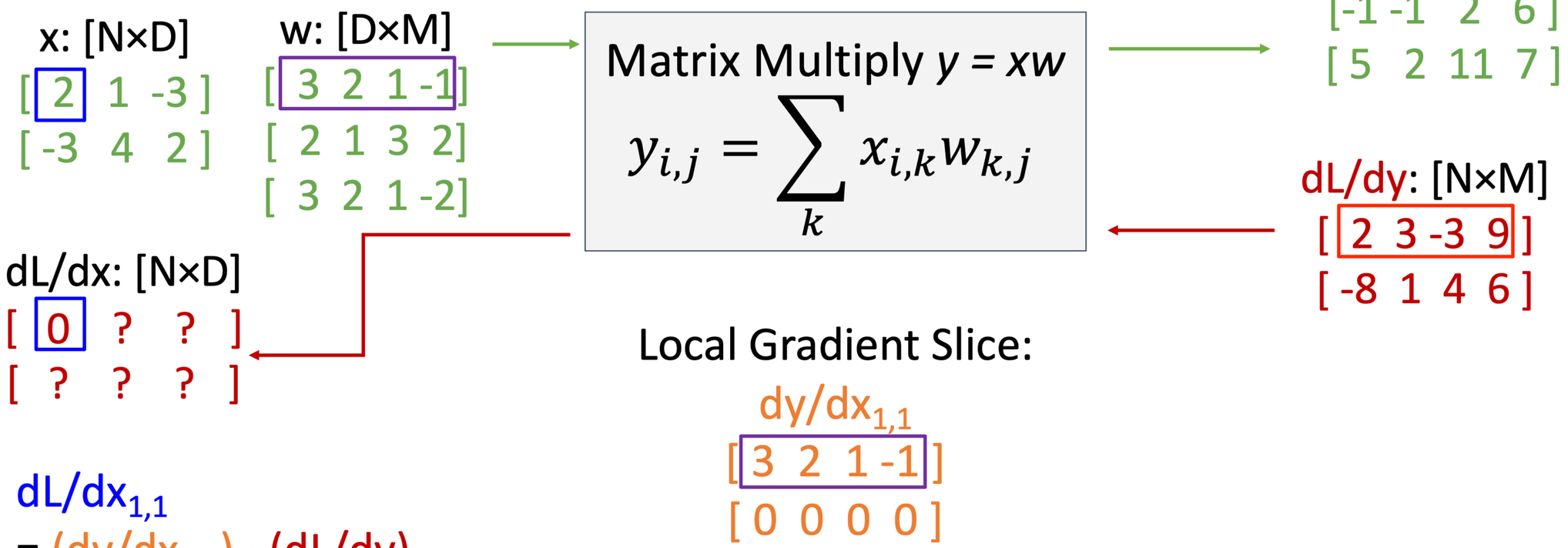
Example: Matrix Multiplication



Example: Matrix Multiplication

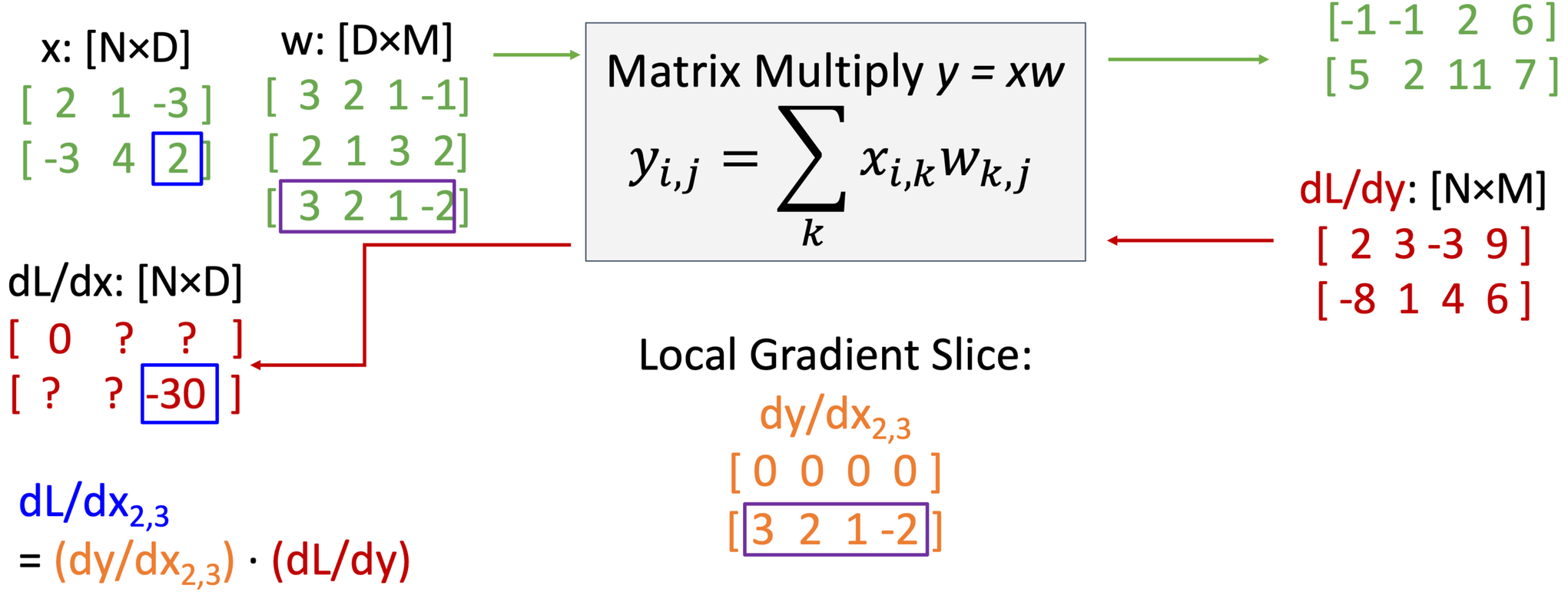


Example: Matrix Multiplication

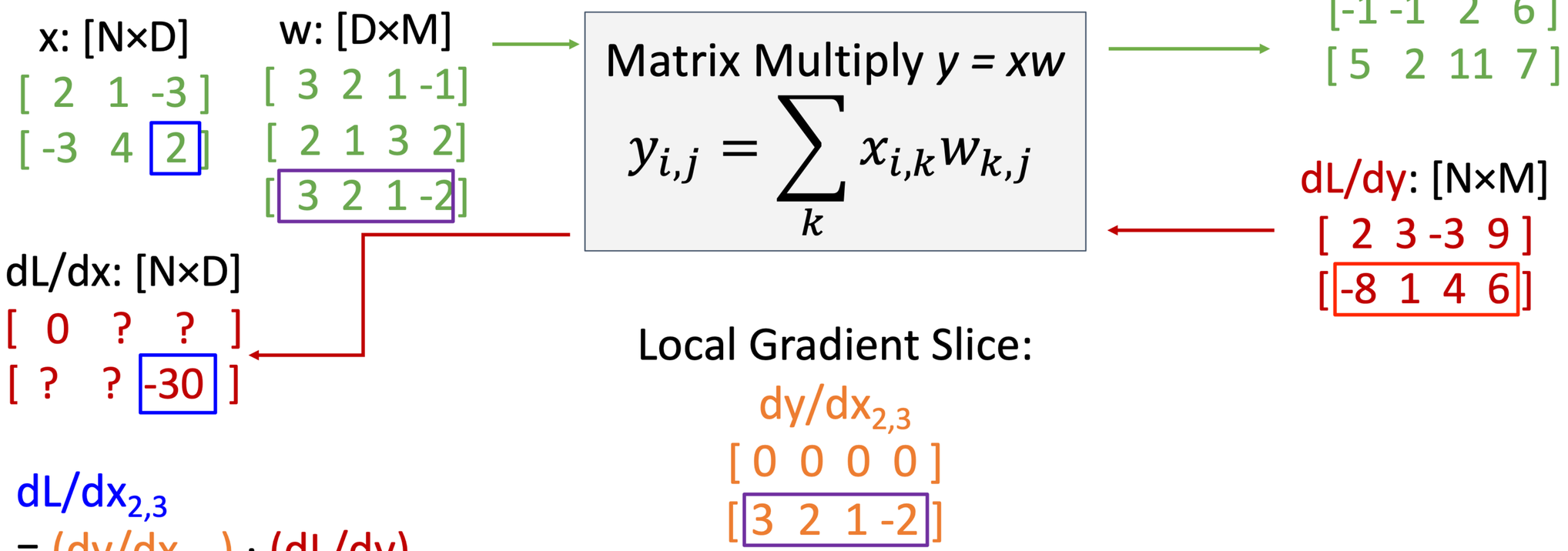


$dL/dx_{1,1}$
$= (dy/dx_{1,1}) \cdot (dL/dy)$
$= (w_{1,:}) \cdot (dL/dy_{1,:})$
$= 3*2 + 2*3 + 1*(-3) + (-1)*9 = 0$

Example: Matrix Multiplication



Example: Matrix Multiplication



$dL/dx_{2,3}$
$= (dy/dx_{2,3}) \cdot (dL/dy)$
$= (w_{3,:}) \cdot (dL/dy_{2,:})$
$= 3*(-8) + 2*1 + 1*4 + (-2)*6 = -30$

Example: Matrix Multiplication

x: [N×D]
[2 1 -3]
[-3 4 2]

w: [D×M]
[3 2 1 -1]
[2 1 3 2]
[3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6]
[5 2 11 7]

dL/dy: [N×M]
[2 3 -3 9]
[-8 1 4 6]

dL/dx: [N×D]
[0 16 -9]
[-24 9 -30]

$dL/dx_{i,j}$
$= (dy/dx_{i,j}) \cdot (dL/dy)$
$= (w_{j,:}) \cdot (dL/dy_{i,:})$

Example: Matrix Multiplication

x: [N×D]
[2 1 -3]
[-3 4 2]

w: [D×M]
[3 2 1 -1]
[2 1 3 2]
[3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6]
[5 2 11 7]

dL/dy: [N×M]
[2 3 -3 9]
[-8 1 4 6]

dL/dx: [N×D]
[0 16 -9]
[-24 9 -30]

$dL/dx = (dL/dy)\, w^T$

[N x D] [N x M] [M x D]

$dL/dx_{i,j}$
$= (dy/dx_{i,j}) \cdot (dL/dy)$
$= (w_{j,:}) \cdot (dL/dy_{i,:})$

Easy way to remember:
It's the only way the shapes work out!

Example: Matrix Multiplication

x: [N×D]
[2 1 -3]
[-3 4 2]

w: [D×M]
[3 2 1 -1]
[2 1 3 2]
[3 2 1 -2]

Matrix Multiply $y = xw$

$$y_{i,j} = \sum_k x_{i,k} w_{k,j}$$

[-1 -1 2 6]
[5 2 11 7]

dL/dy: [N×M]
[2 3 -3 9]
[-8 1 4 6]

dL/dx: [N×D]
[0 16 -9]
[-24 9 -30]

$dL/dx = (dL/dy)\, w^T$

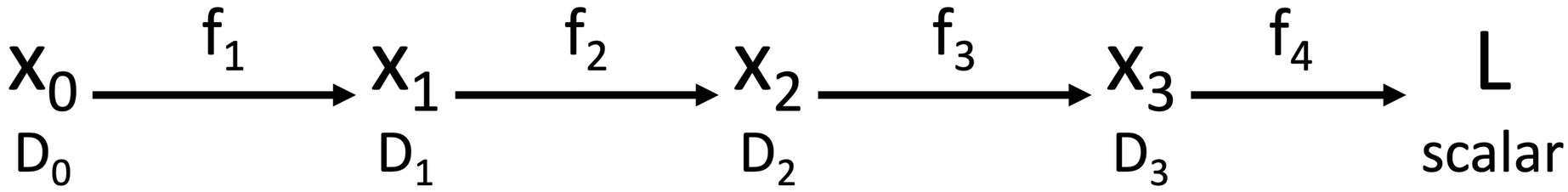
[N x D] [N x M] [M x D]

$dL/dw = x^T\, (dL/dy)$

[D x M] [D x N] [N x M]

Easy way to remember:
It's the only way the shapes work out!

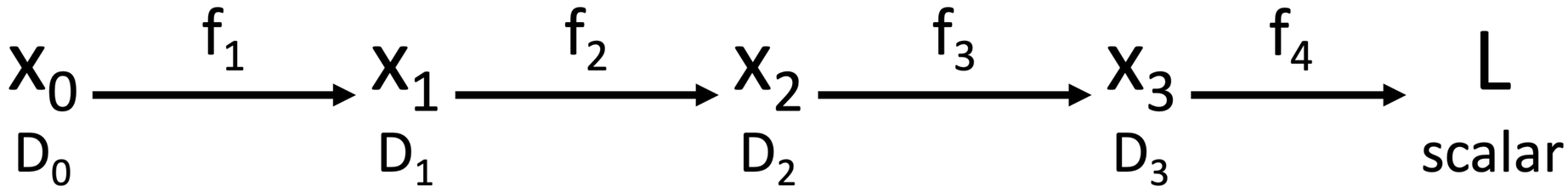
Backpropagation: Another View



Chain rule

$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)\left(\frac{\partial L}{\partial x_3}\right)$$

Backpropagation: Another View



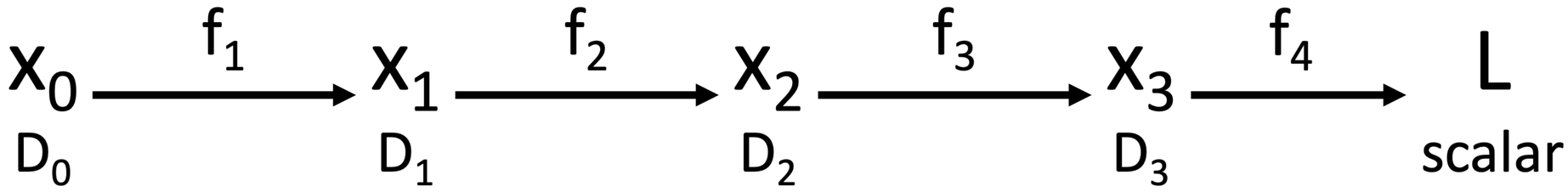
Matrix multiplication is **associative**: we can compute products in any order

Chain rule

$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)\left(\frac{\partial L}{\partial x_3}\right)$$

$[D_0 \times D_1]$ $[D_1 \times D_2]$ $[D_2 \times D_3]$ $[D_3]$

Reverse-Mode Automatic Differentiation



Matrix multiplication is **associative**: we can compute products in any order

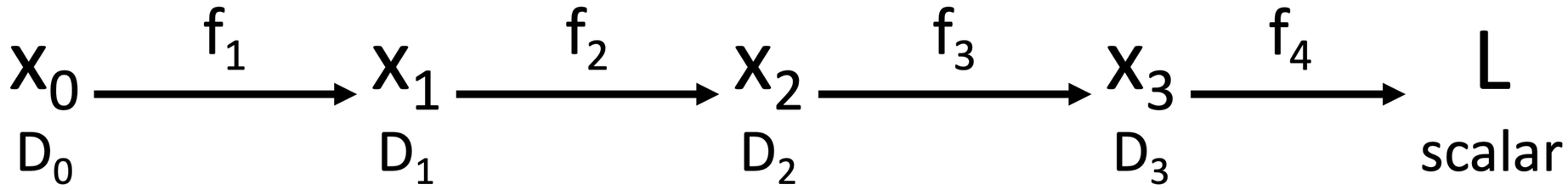
Computing products right-to-left avoids matrix-matrix products; only needs matrix-vector

Chain rule

$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)\left(\frac{\partial L}{\partial x_3}\right)$$

$[D_0 \times D_1]$ $[D_1 \times D_2]$ $[D_2 \times D_3]$ $[D_3]$

Reverse-Mode Automatic Differentiation



Matrix multiplication is **associative**: we can compute products in any order

Computing products right-to-left avoids matrix-matrix products; only needs matrix-vector

Chain rule

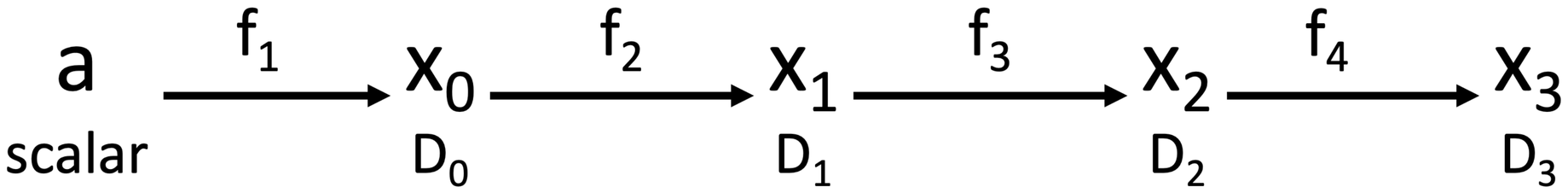
$$\frac{\partial L}{\partial x_0} = \left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)\left(\frac{\partial L}{\partial x_3}\right)$$

$[D_0 \times D_1]$ $[D_1 \times D_2]$ $[D_2 \times D_3]$ $[D_3]$

Compute grad of scalar output w/respect to all vector inputs

What if we want grads of scalar input w/respect to vector outputs?

Forward-Mode Automatic Differentiation

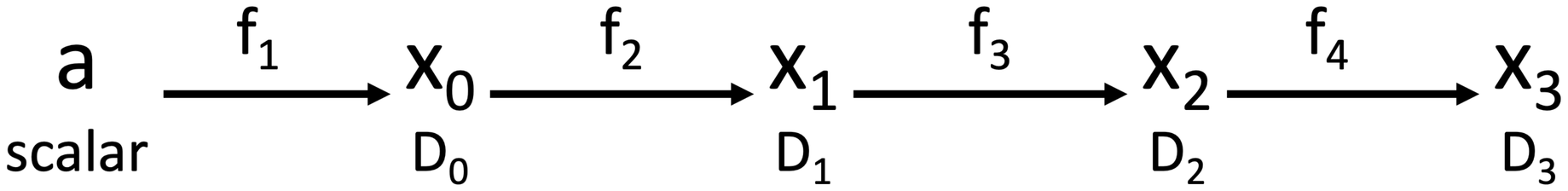


Chain rule

$$\frac{\partial x_3}{\partial a} = \left(\frac{\partial x_0}{\partial a}\right)\left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)$$

$[D_0]$ $[D_0 \times D_1]$ $[D_1 \times D_2]$ $[D_2 \times D_3]$

Forward-Mode Automatic Differentiation



Computing products left-to-right avoids matrix-matrix products; only needs matrix-vector

Chain rule

$$\frac{\partial x_3}{\partial a} = \left(\frac{\partial x_0}{\partial a}\right)\left(\frac{\partial x_1}{\partial x_0}\right)\left(\frac{\partial x_2}{\partial x_1}\right)\left(\frac{\partial x_3}{\partial x_2}\right)$$

$[D_0]$ $[D_0 \times D_1]$ $[D_1 \times D_2]$ $[D_2 \times D_3]$

Universal Approximation

A neural network with one hidden layer can approximate any function $f: \mathbb{R}^N \rightarrow \mathbb{R}^M$ with arbitrary precision*

*Many technical conditions: Only holds on compact subsets of $\mathbb{R}^N$; function must be continuous; need to define "arbitrary precision"; etc.

Universal Approximation

Let $\phi(\cdot)$ be a non-constant, bounded and monotone-increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, there exists an integer m_1 and sets of real constants α_i, β_i, and w_{ij}, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \phi \left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right)$$

as an approximation realization of the function $f(\cdot)$: that is,

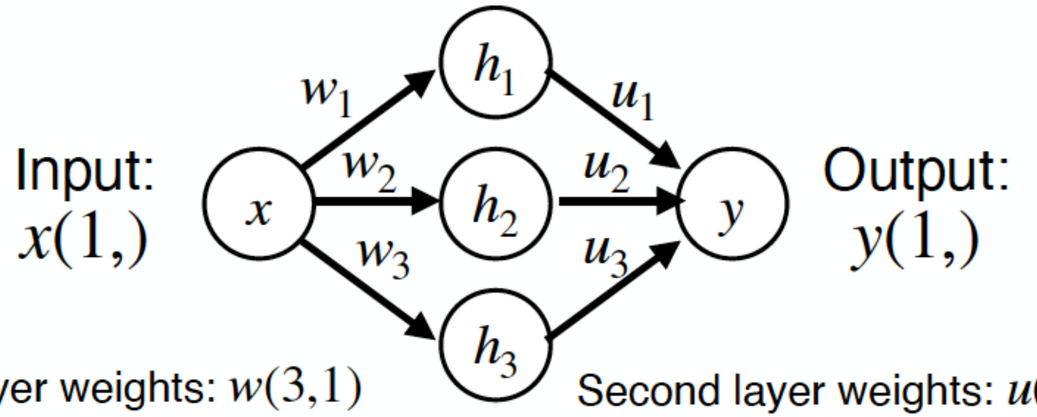
$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \epsilon$$

for all $x_1, x_2, \ldots, x_{m_0}$ in the input space.

- Essentially, the Universal Approximation Theorem states that a single hidden layer is sufficient for a multilayer perceptron to compute a uniform ϵ approximation to a given training set - provided you have the *right* number of neurons and the *right* activation function.
  (However, this does not say that a single hidden layer is optimal with regards to learning time, generalization, etc.)

Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network



First layer weights: $w(3,1)$

First layer bias: $b(3,)$

Second layer weights: $u(3,1)$

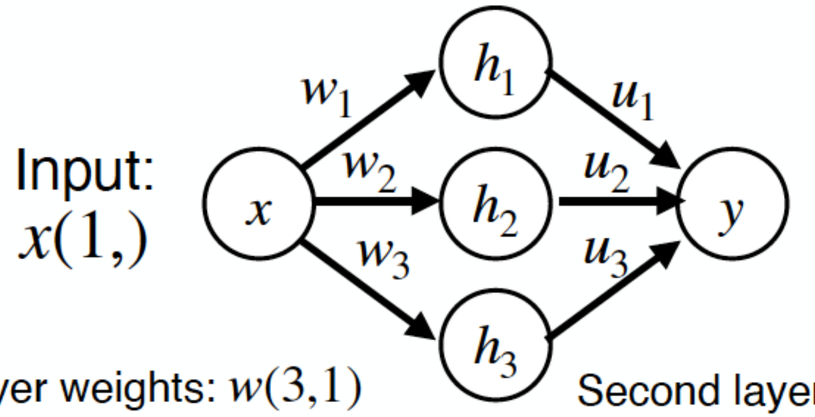
First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1) + u_2 \max(0, w_2 x + b_2) + u_3 \max(0, w_3 x + b_3) + p$$

Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network



First layer weights: $w(3,1)$

First layer bias: $b(3,)$

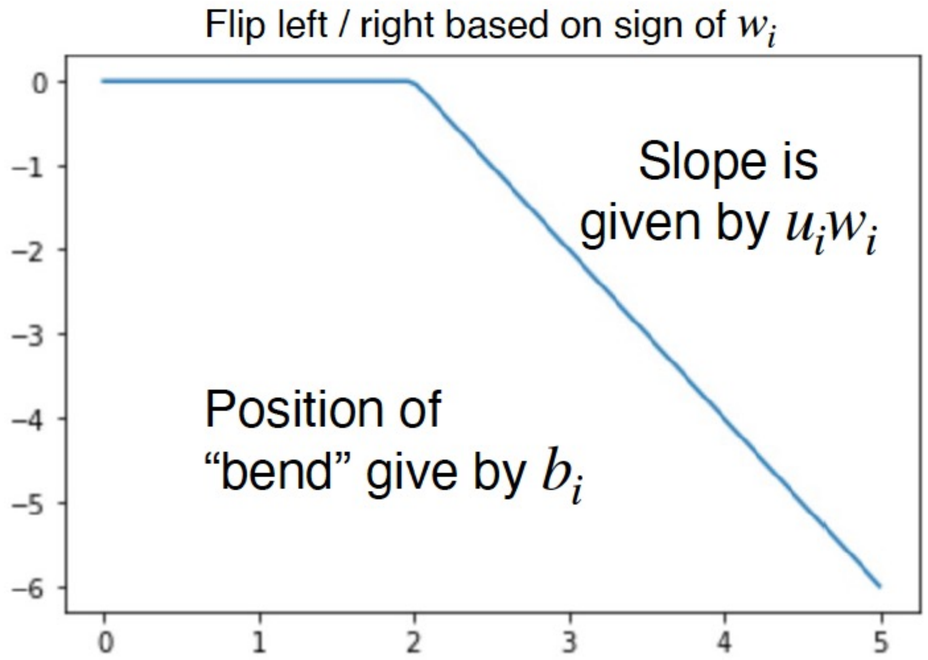
Second layer weights: $u(3,1$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

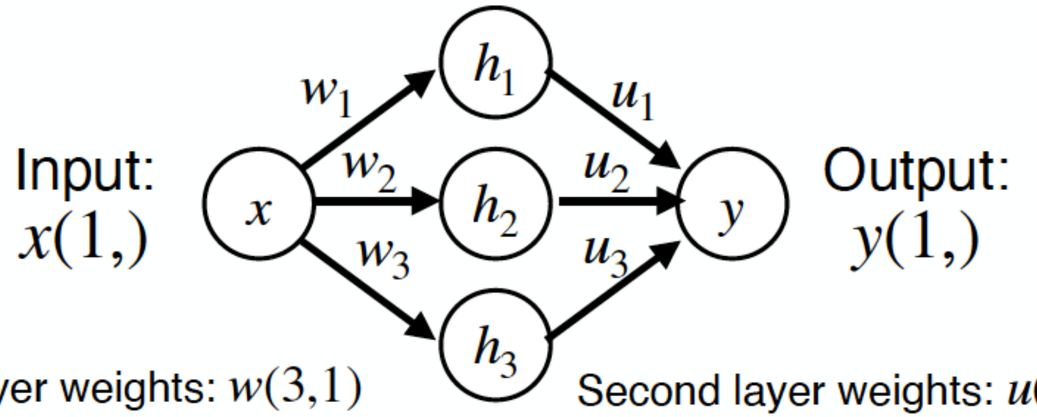
$$y = u_1 \max(0, w_1 x + b_1) + u_2 \max(0, w_2 x + b_2) + u_3 \max(0, w_3 x + b_3) + p$$

Output is a sum of shifted, scaled ReLUs:



Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network



First layer weights: $w(3,1)$

First layer bias: $b(3,)$

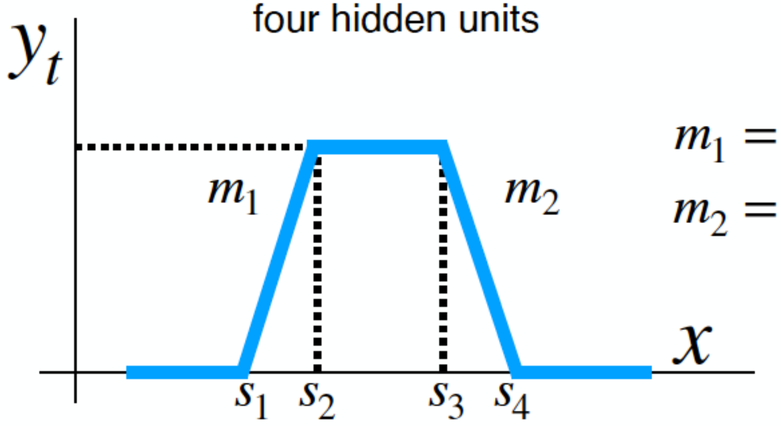
Second layer weights: $u(3,1)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1) + u_2 \max(0, w_2 x + b_2) + u_3 \max(0, w_3 x + b_3) + p$$

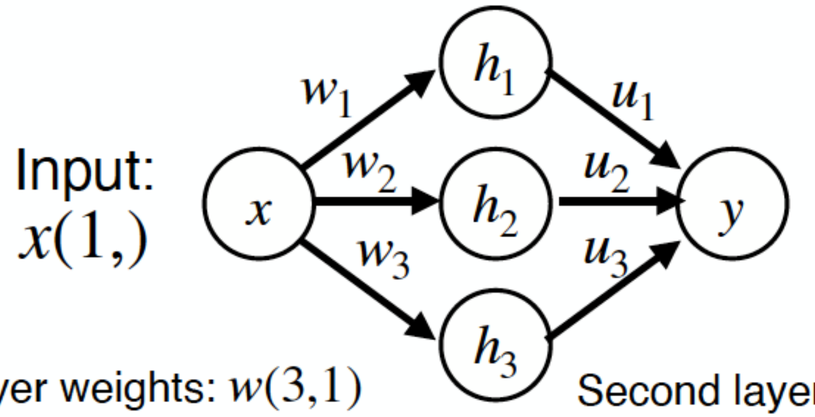
We can build a "bump function" using four hidden units



$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$

Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network



First layer weights: $w(3,1)$

First layer bias: $b(3,)$

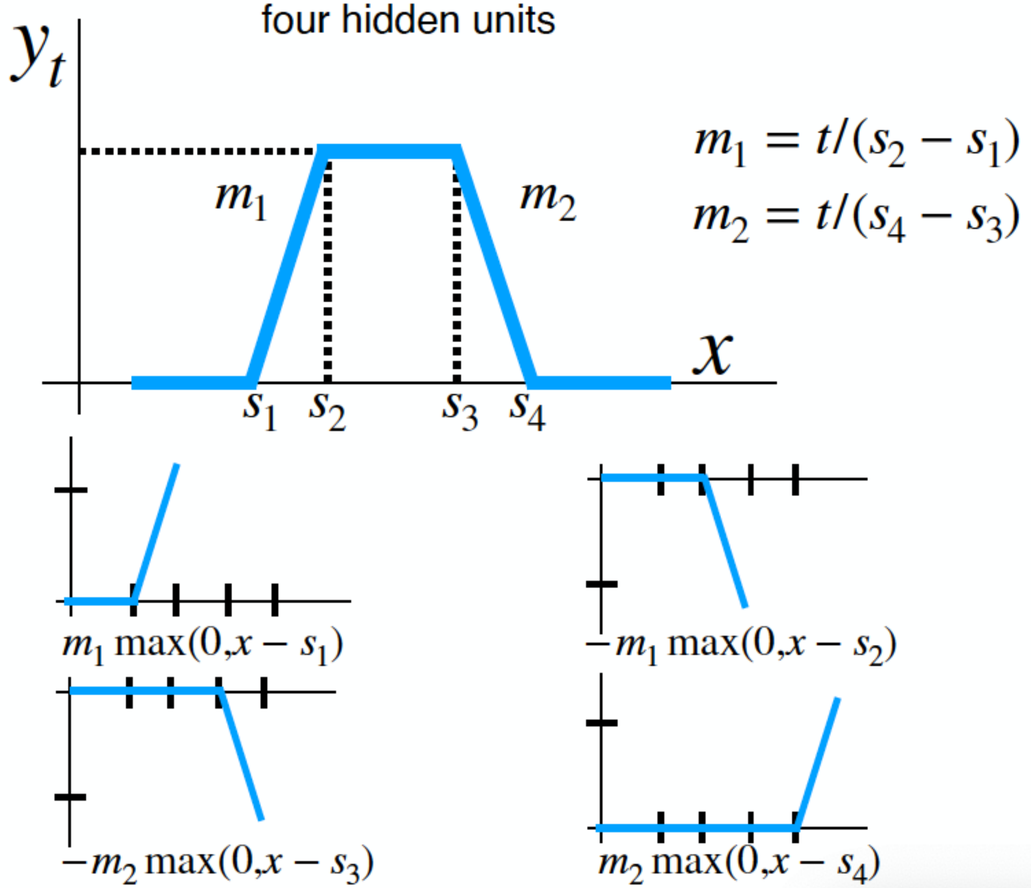
Second layer weights: $u(3,1)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

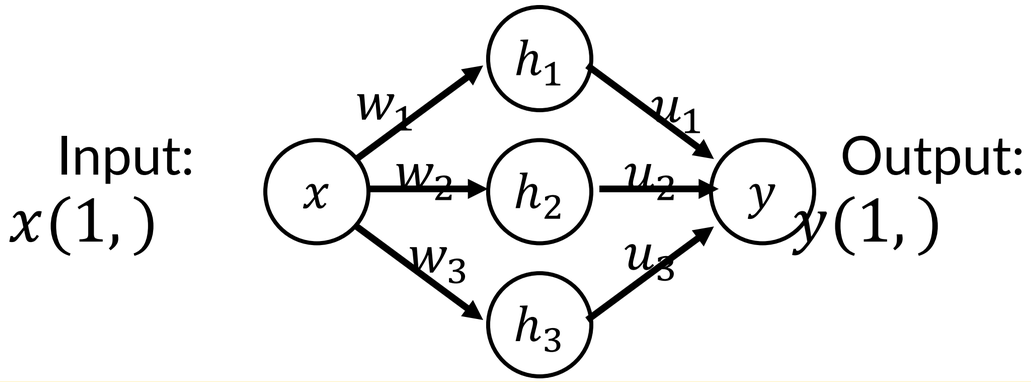
$$y = u_1 \max(0, w_1 x + b_1) + u_2 \max(0, w_2 x + b_2) + u_3 \max(0, w_3 x + b_3) + p$$

We can build a "bump function" using four hidden units



Universal Approximation

Example: Approximating a function $f: \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network



Universal approximation tells us:

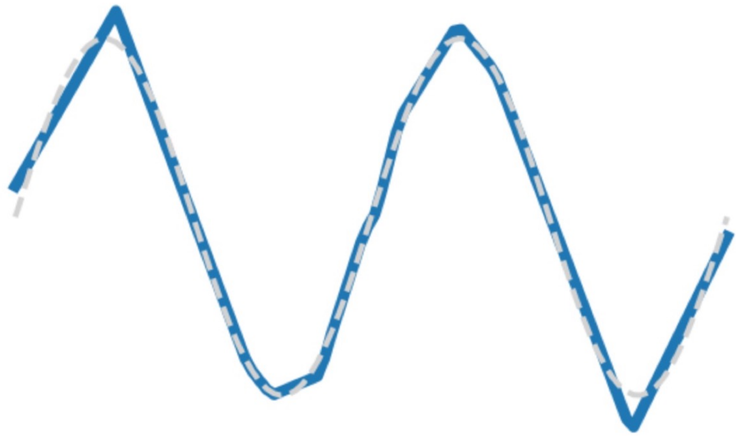
- Neural nets can represent any function

Universal approximation DOES NOT tell us:

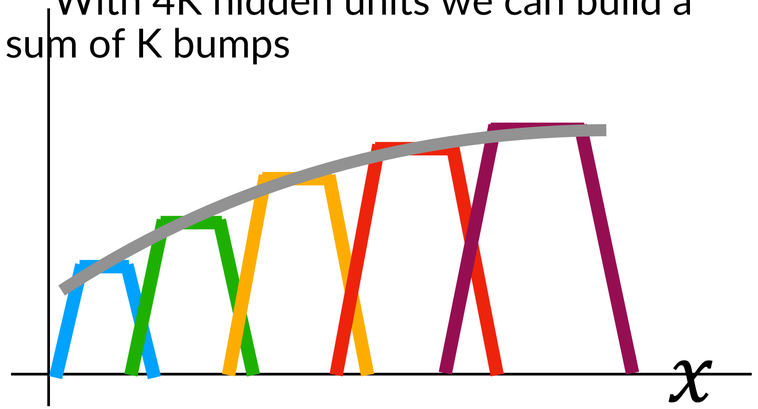
- Whether we can actually learn any function with SGD
- How much data we need to learn a function

Remember: kNN is also a universal approximator!

Reality check: Networks don't really learn bumps!



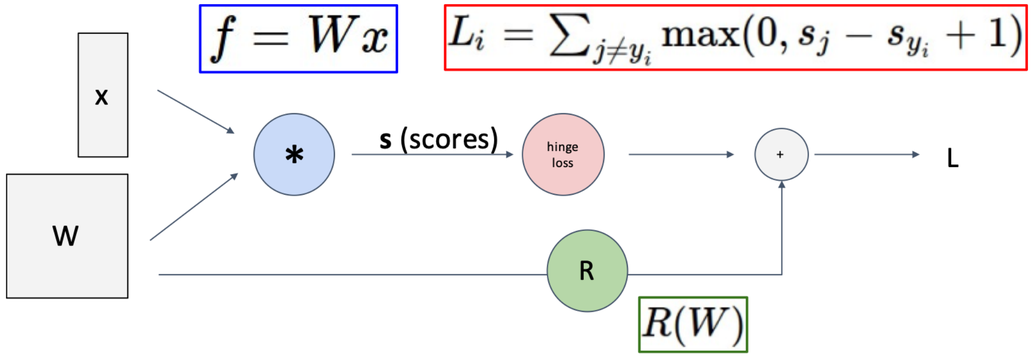
With 4K hidden units we can build a sum of K bumps



Approximate functions with bumps!

Summary

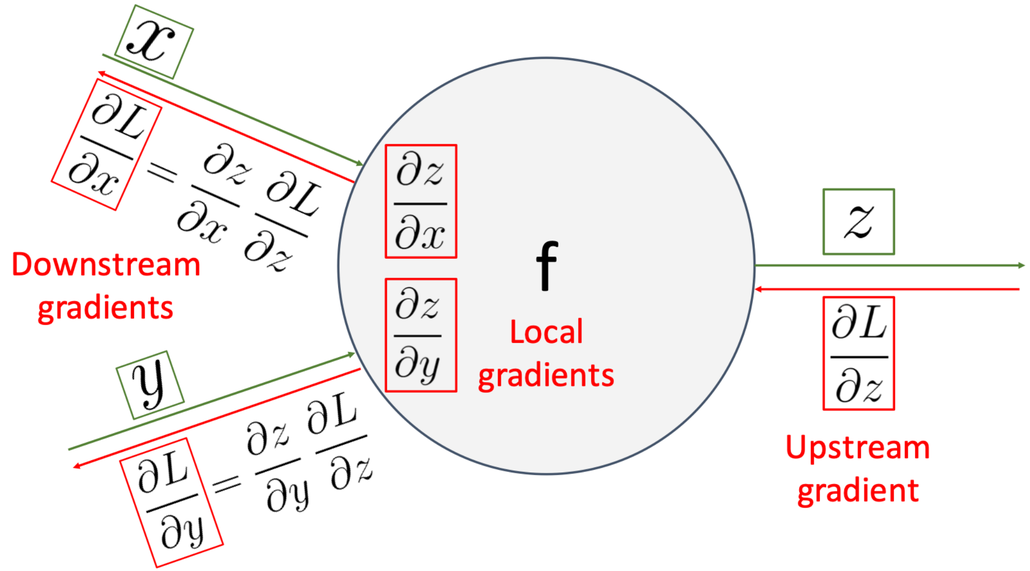
Represent complex expressions as **computational graphs**



Forward pass computes outputs

Backward pass computes gradients

During the backward pass, each node in the graph receives **upstream gradients** and multiplies them by **local gradients** to compute **downstream gradients**



Summary

Backprop can be implemented with “flat” code where the backward pass looks like forward pass reversed (Use this for A2!)

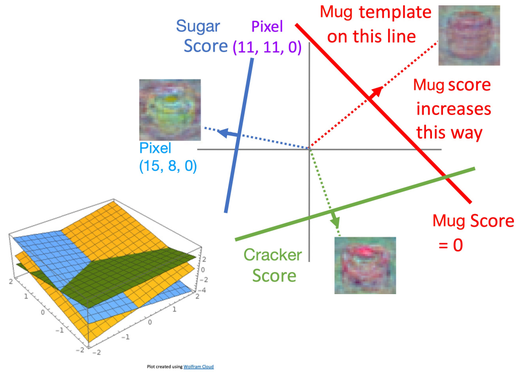
```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)

  grad_L = 1.0
  grad_s3 = grad_L * (1 - L) * L
  grad_w2 = grad_s3
  grad_s2 = grad_s3
  grad_s0 = grad_s2
  grad_s1 = grad_s2
  grad_w1 = grad_s1 * x1
  grad_x1 = grad_s1 * w1
  grad_w0 = grad_s0 * x0
  grad_x0 = grad_s0 * w0
```

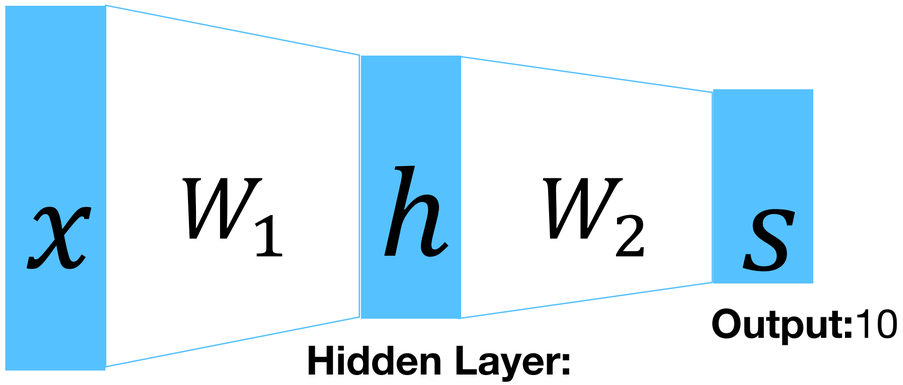
Backprop can be implemented with a modular API, as a set of paired forward/backward functions

```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Summary



$$f(x) = W_2 max(0, W_1 x + b_1) + b_2$$

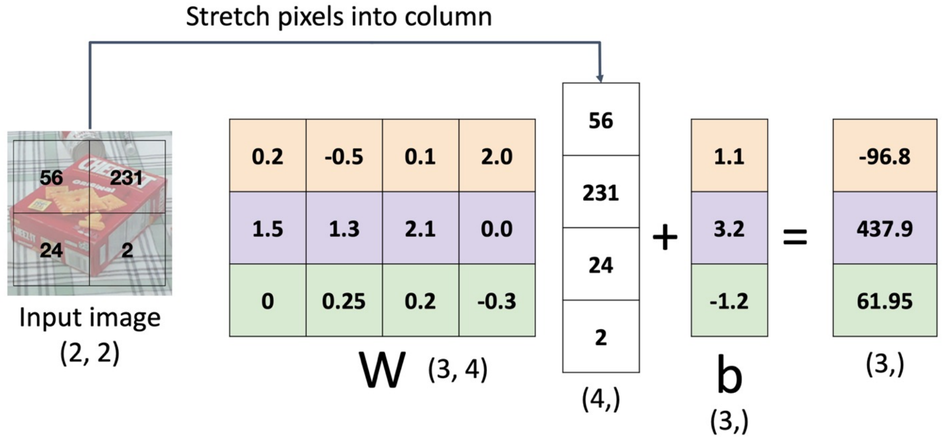


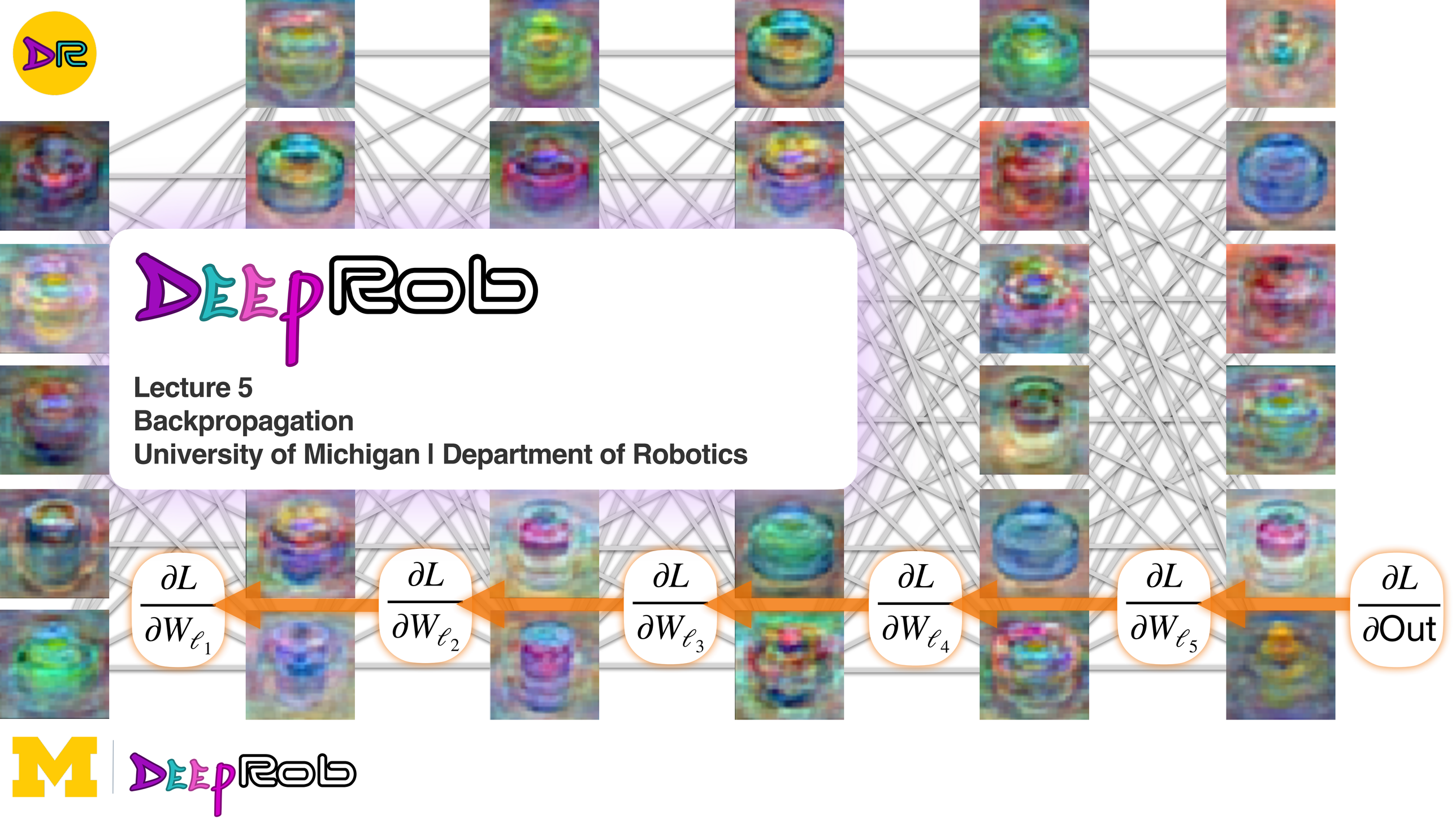
Input:
3072

Hidden Layer:
100

Output:10

Problem: So far our classifiers don't respect the spatial structure of images!




DEEPROB
Lecture 5
Backpropagation
University of Michigan | Department of Robotics
$\frac{\partial L}{\partial W_{\ell_1}}$
$\frac{\partial L}{\partial W_{\ell_2}}$
$\frac{\partial L}{\partial W_{\ell_3}}$
$\frac{\partial L}{\partial W_{\ell_4}}$
$\frac{\partial L}{\partial W_{\ell_5}}$
$\frac{\partial L}{\partial \text{Out}}$
DEEPROB